Deixe seu código limpo e brilhante

Desmistificando Clean Code com Java e Python

Casa do Código | alura

JOSÉ YOSHIRIRO

Sumário

- ISBN
- Agradecimentos
- Sobre o livro
- Profissionais que colaboraram nesta obra
- Prefácio
- Comentários sobre o livro
- 1. O que é Código Limpo
- 2. Bons nomes
- 3. Funções
- 4. Comentários de código
- 5. Formatação de código
- 6. Objetos
- 7. Tratamento de erros
- 8. Testes de unidade (ou unitários)
- 9. Classes
- 10. A regra do escoteiro
- 11. Apêndice A — Convenções de código
- 12. Apêndice B — Ferramentas
- 13. Referências

ISBN

Impresso: 978-85-5519-337-8

Digital: 978-85-5519-336-1

Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

Yoshiriro, José
Deixe seu código limpo e brilhante : desmistificando Clean Code com Java e Python / José Yoshiriro. -- São Paulo, SP : Aovs Sistemas de Informática, 2023.

Bibliografia.
ISBN 978-85-5519-337-8

1. Java (Linguagem de programação) 2. Programação (Computadores) 3. Python (Linguagem de programação) 4. Software - Desenvolvimento I. Título.

23-153025 CDD-005.1

Índices para catálogo sistemático:

1. Programação de computadores 005.1

Tábata Alves da Silva - Bibliotecária - CRB-8/9253

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

Agradecimentos

Agradeço primeiramente a Deus, pelo dom da vida, por todas as oportunidades que permitiu que eu tivesse e por todos os livramentos que sempre me proporcionou.

Agradeço a minha mãe, dona Mizuko, que sempre fez de tudo para que eu tivesse uma educação de qualidade ao alcance de suas possibilidades financeiras. Sua insistência no meu foco nos estudos foi determinante para que eu me tornasse o profissional que sou.

Agradeço a meu pai, José (*In Memoriam*), que deu todo o apoio para que eu somente estudasse durante a infância e adolescência.

Agradeço a minha amada esposa e companheira de muitos anos, Danusa. Seu jeito carinhoso e meigo são uma contínua inspiração para meu cotidiano pessoal e profissional. Ela me deu um filho maravilhoso, o Tadashi, a quem amo sem medida.

Agradeço ao professor Eduardo Martins Guerra, que foi o primeiro a publicar meus textos, em forma de artigos para a antiga revista MundoJava, tornando-me conhecido nacionalmente. Suas revisões e dicas certamente me fizeram um escritor melhor.

Agradeço aos profissionais que deram suas opiniões sobre as técnicas de código limpo. São eles: Eduardo Guerra, Camila Achutti, Rafael Oliveira, Renan Rodrigo e Rodrigo Vieira.

E agradeço a você que adquiriu esta obra. Espero que o livro ajude você a escrever códigos limpos, brilhantes e incríveis!

Sobre o livro

Este livro é uma coleção de códigos de exemplos em Java e Python voltados para iniciantes em programação visando explorar os principais conceitos abordados na obra ***Clean Code***, escrita por *Robert Cecil Martin* (conhecido como *uncle Bob*), em 2008.

Esta obra **não é uma tradução**. Os principais ensinamentos do livro *Clean Code* são citados aqui com outras palavras, com linguagem e exemplos que espero que sejam mais fáceis de serem aprendidos por iniciantes em desenvolvimento de software.

Sou professor há mais de uma década e perdi as contas de quantas vezes indiquei o *Clean Code* para meus alunos. Então notei que os mais iniciantes davam um *feedback* um tanto negativo. Não sobre a obra, mas sobre a leitura. É que era comum me falarem que não conseguiam compreender boa parte dos conceitos devido aos termos técnicos e exemplos de código contidos no livro. Outros não tinham acesso à versão traduzida e não conseguiam ler em inglês. Outros ainda só tinham acesso à versão traduzida e acharam ela confusa (pois os códigos de exemplo são em inglês nela).

Cheguei então à conclusão de que, para iniciantes em programação e/ou pessoas sem proficiência de leitura em inglês, o livro tinha várias barreiras. Por isso decidi escrever esta obra toda em português e com linguagem e exemplos de códigos mais simples para iniciantes. Também resolvi dar exemplos de códigos em duas linguagens de programação muito populares (**Java** e **Python**) para poder tornar o conteúdo mais acessível para uma maior quantidade de estudantes iniciantes no mundo do desenvolvimento de software. A versão do Java usada foi a **21**. A versão do Python usada foi a **3.13**.

Nem todos os conceitos da obra que inspirou este livro estão aqui. Achei melhor não citar aqueles que acredito que não são apropriados para serem ensinados ao público-alvo desta obra

(novatos e novatas em programação). Por favor, *uncle Bob*, não fique chateado comigo ;)

Público-alvo

Esta obra destina-se a estudantes e profissionais iniciantes de desenvolvimento de software. Apesar de a obra conter exemplos de código em Java e Python, estudantes e profissionais de várias outras linguagens podem aprender com ela, contanto que a sintaxe e as convenções de código não sejam muito diferentes às dessas duas linguagens.

Este livro NÃO é uma obra para ensinar a programar; porém, se propõe a ser uma ferramenta de aprofundamento para quem ainda é iniciante nesse mundo do desenvolvimento de software.

Pré-requisitos

Para melhor acompanhar o conteúdo desta obra, basta ter noções básicas de programação (variáveis, funções e estruturas de controle e repetição) e pelo menos uma pequena noção dos conceitos da programação orientada a objetos.

Sobre o autor

José Yoshiriro Ajisaka Ramos, mais conhecido como **Yoshi**, nasceu em 1981 em Belém (PA). É bacharel em Sistemas de Informação, especialista em Engenharia de Sistemas e mestre em Administração das Organizações pela USP. Possui certificações internacionais OCUP

Fundamental, SCBCD 5, SCWCD 5, SCJP 6 e SOA Suite 11g Certified Implementation Specialist.

Atua profissionalmente com desenvolvimento de software desde 2002, mas programa desde 1994. Atuou na criação e/ou manutenção de sistemas usados em empresas como TJE-PA, BrasilPrev, Rico, Caixa Econômica Federal, DETRAN-PA, BASA, Citibank, American Red Cross (EUA) e OneBlood (EUA). Também colaborou com os projetos open source Struts 2, Grails e Hibernate Validator. Programar é uma de suas grandes paixões.

É professor desde 2003, atuando como instrutor em vários cursos de programação. Como docente em nível superior, já trabalhou para: UAB/IFPA; Faculdade Ipiranga e FIAP. No início de 2014, passou a compor o corpo docente da faculdade São Paulo Tech School - SPTech. Certamente já teve mais de mil alunos. Publicou vários artigos sobre programação nas revistas Mundo Java e GroovyMag (EUA). É autor de outro livro, o *Spock framework - Testes automatizados ágeis para Java e REST* publicado pela Casa do Código. Palestrou em diversos eventos de tecnologia, como na Virada Tecnológica de São Paulo 2017, nos TDC São Paulo de 2014, 2015, 2017 e 2019, e no FISL de 2015. Ensinar é sua grande paixão.

LinkedIn: https://linkedin.com/in/jyoshiriro

Profissionais que colaboraram nesta obra

Alguns incríveis profissionais de desenvolvimento de software ajudaram gentilmente na construção desta obra. Eles deram suas valiosas opiniões sobre os princípios de código limpo. Essas opiniões estão distribuídas por todo o livro. Foram eles:

Eduardo Martins Guerra

Doutor e mestre em Engenharia Eletrônica e Computação pelo ITA. Atualmente faz pesquisa na área de engenharia de software na Free University of Bolzen-Bolzano. Foi editor da revista Mundo Java por 7 anos, atuou como professor no ITA e como pesquisador no INPE. Possui mais de 100 artigos técnicos publicados em conferências e revistas nacionais e internacionais.

LinkedIn: https://linkedin.com/in/eduardo-guerra-b4633115b

Camila Fernandez Achutti

Mestra em Ciência da Computação pela USP. Doutoranda pela Escola Politécnica da USP. Trabalha com desenvolvimento de software desde 2011. Tem as seguintes empresas em seu currículo: Mastertech, SOMAS, Insper, FIAP e Google.

LinkedIn: https://linkedin.com/in/camilaachutti

Rafael Santana Oliveira

Mestre em Ciência da Computação pela UFPA. Trabalha com desenvolvimento de software desde 2013. Tem as seguintes empresas em seu currículo: Metasix, Itaú e Zettle by PayPal.

LinkedIn: https://linkedin.com/in/rafael-santana-oliveira

Renan Rodrigo Barbosa

Mestre em Ciência da Computação pelo IME-USP. Trabalha com desenvolvimento de software desde 2017. Tem as seguintes empresas em seu currículo: SENAI, UNESP, Mastertech, Red Hat e Canonical.

LinkedIn: https://linkedin.com/in/renangerbilo

Rodrigo Vieira Pinto

Mestre em Engenharia de Software pelo IPT-SP e especialista em Engenharia de Software pela PUC-SP. Trabalha com desenvolvimento de software desde 2004. Tem as seguintes empresas em seu currículo: Elo7, PicPay, Tokio Marine, SPTech (antiga Bandtec), VR Benefícios, Caelum, Alura e FIAP.

LinkedIn: https://linkedin.com/in/rodrigovp

Prefácio

Por Prof. MSc. Carlos Rafael Gimenes das Neves

Acabaram de se completar 27 anos desde que iniciei no mundo da codificação de maneira não profissional, aos 12 anos de idade. E foi aos 16 anos de idade que, como parte do meu estágio obrigatório por estar fazendo "colegial técnico", atuei profissionalmente pela primeira vez, desenvolvendo sites para a empresa do meu professor, que me convidou para trabalhar logo que teve a oportunidade ao longo do 3º ano do ensino médio.

Desde aquela época, experimentei na prática a ascensão e queda de diversas linguagens de programação, plataformas, ferramentas, frameworks, bibliotecas e o que mais se possa imaginar. Olhar para trás nesse momento é um exercício que por vezes desperta um certo sorriso no rosto, o mesmo sorriso que você dá quando vê seu filho tentando fazer algo pela primeira vez, que você sabe que vai dar errado, mas o deixa tentar mesmo assim, só para que aprenda. Digo isso porque era comum ouvir frases como "*se você não souber X, nunca conseguirá coisa alguma na vida*". Ora, onde está esse X hoje, quer seja X uma linguagem de programação ou uma ferramenta qualquer?

Alguns desses X que foram concebidos ao longo dos anos 1980, 1990 e até mesmo 2000 ainda estão por aí, firmes e fortes, sustentando empresas e, direta ou indiretamente, auxiliando no nosso dia a dia. Porém, incontáveis X dessa mesma época hoje descansam em paz no cemitério virtual das tecnologias passadas.

Fiz esse retorno ao passado apenas para ajudar a enaltecer um ponto importante: diferente de inúmeras ferramentas tecnológicas esquecidas, algumas coisas permaneceram na minha vida todo esse tempo, e só têm evoluído, graças a Deus: a habilidade de resolver

problemas, a criatividade para combinar diferentes soluções, a lógica além de tantos outras *soft skills*.

Quem me conhece sabe que eu canto aos sete ventos: "*não se deve idolatrar a chave de fenda*". Ferramentas passam. As pessoas e suas habilidades ficam. Não importa se eu preciso escrever ABC em uma linguagem de programação para ordenar uma lista de maneira crescente, enquanto em outra linguagem eu precise escrever XYZ. Isso não torna uma linguagem melhor ou pior do que a outra. O fato é que eu quero ordenar uma lista! Esse tipo de necessidade — a de expressar uma ideia de modo a resolver um problema — não mudou ao longo dos anos.

Assim como ocorre com linguagens humanas, como Português ou Inglês, as linguagens de programação oferecem ao escritor (programador) uma liberdade quase poética para expressar suas ideias da forma que quiser. Em português existem textos sucintos, prolixos, confusos, limpos... Alguns agradam o leitor no momento da leitura, enquanto outros, por vezes, sequer conseguem transmitir claramente a ideia original de seu escritor. Junto à liberdade poética na hora de se expressar, as linguagens de programação também trazem consigo a característica de permitir a existência de textos (códigos) que agradarão seu leitor, bem como textos que não conseguirão transmitir claramente a ideia original.

O professor Yoshiriro foi muito feliz na escolha do tema deste livro. Quem programa provavelmente já ouviu expressões como "*Compilou? Então funciona!*", ou já se deparou com incríveis soluções técnicas paliativas temporárias, elaboradas pelas mentes de profissionais apenas ansiando irem para casa em uma sexta-feira no final da tarde. O fato é que nem sempre algo que compila efetivamente funciona. Nem sempre uma solução temporária de uma sexta-feira de tarde desaparece na próxima segunda-feira de manhã. É muito fácil identificar e criticar códigos dessa natureza, mesmo que eles sejam capazes de transmitir alguma ideia ao leitor. Por outro lado, nem todo código "que compila", que funciona corretamente e que não foi produto de uma solução temporária em

uma sexta-feira de tarde é capaz de transmitir aos futuros leitores a ideia brilhante contida nele.

O desejo do Yoshi de tirar este assunto da esfera ocupada por profissionais experientes, aliado à forma como ele aborda o tema e expressa suas com clareza e sem mistério, faz desta obra um excelente guia para que pessoas de todos os níveis de proficiência consigam expressar suas ideias e deixar um legado limpo e compreensível para os que precisarão ler suas criações.

Boa leitura, pessoal! E boa sorte na jornada perene em busca por aperfeiçoamento das habilidades de conversão de ideias brilhantes em códigos mais brilhantes ainda!

— **Carlos Rafael Gimenes das Neves** é mestre em Engenharia pelo ITA-SP. Trabalha com desenvolvimento de software desde 1999 e já trabalhou para as empresas Debis Humaitá (atual T-Systems), Compugraf, Olimpus Automotive, Magna Sistemas, até ter se tornado diretor de tecnologia em sua empresa, a Smooh. É professor desde 2007, tendo previamente ministrado aulas na FMU, FAENAC, BandTec (atual SPTech) e, atualmente, na ESPM.

LinkedIn: https://linkedin.com/in/carlosrafaelgn

Comentários sobre o livro

"*Um dos pontos que me chamou atenção logo no início da obra é que, diferente de nadar para sobreviver, programar frequentemente envolverá consertar, melhorar ou ampliar o código. É nesse contexto que este livro apresenta as orientações de código limpo como um conjunto de práticas de uma arte marcial que podem ser adotadas conforme o estilo ou necessidade de cada lutador.*

Além disso, a obra traz exemplos em Java e em Python, conferindo um diferencial substancial, porque o Python é considerado uma linguagem com sintaxe e compreensão acessíveis, acompanhada de didática facilmente absorvida por pessoas que estão iniciando na área. A obra reúne exemplos cotidianos, pontos de vista de profissionais acerca das práticas, perguntas que estimulam a imersão do leitor nos exemplos citados em uma leitura muito fluida, divertida e com ótimas analogias."

— **Giovana Miniguiti**, bacharel em Gestão Ambiental pela USP. Iniciou a transição de carreira em 2022, quando iniciou no curso de Análise e Desenvolvimento de Sistemas pela SPTech.

LinkedIn: https://linkedin.com/in/giovanna-miniguiti-497885139

"*Adorei a abordagem do Yoshi para falar sobre código limpo, porque possui exemplos e uma linguagem superacessíveis não só para*

iniciantes, mas para os experientes que não tiveram contato com o assunto também. Com certeza tudo seria diferente se eu e outros colegas devs tivéssemos acesso a um conteúdo tão rico e simples como este no início da carreira. Recomendo facilmente este livro para os devs que lidero."

— **Emanuel de Sousa Oliveira**, tecnólogo em Análise e Desenvolvimento de Sistemas pela SPTech. Trabalha com desenvolvimento de software desde 2018 e tem as seguintes empresas no currículo: TIVIT, Hands, KaBuM! e Grupo Data.

LinkedIn: https://linkedin.com/in/gtmany

"*A obra do professor Yoshi é muito oportuna e importante, pois é um material essencial para os novos desenvolvedores de software e, ao mesmo tempo, um excelente material de referência e reflexão para desenvolvedores mais experientes na produção e manutenção de programas que guardam os conceitos de código limpo.*

Sua obra trata os principais pontos para que se produza um código limpo. Foi escrita em uma linguagem simples, com exemplos objetivos e divertidos, facilitando sua leitura e compreensão, mas sem diminuir a profundidade dos assuntos tratados. Os códigos não limpos consomem muito mais tempo para a árdua fase de manutenção de software. Na minha opinião, esse conceito deveria ser adotado por todos os desenvolvedores e suas equipes."

— **Cláudio Frizzarini (Frizza)**, mestre em Sistemas de Informação pela USP e especialista em Gestão Estratégica pela FGV. Trabalha com desenvolvimento de software desde 1982 e tem as seguintes empresas no currículo: Prodesp, Agroceres, Universidade São Judas Tadeu, Copersucar e SPTech.

LinkedIn: https://linkedin.com/in/claudio-frizzarini

Capítulo 1
O que é Código Limpo

Há décadas, áreas do conhecimento como administração de empresas e engenharias seguem normas do tipo ISO ou orientações como o 5S. Profissionais da área de saúde seguem um POP (Procedimento Operacional Padrão) de acordo com a ação a ser realizada. Já no desenvolvimento de software, até o início dos anos 2000, apesar de haver padronizações no processo geral de desenvolvimento, como CMMI, MPS.BR, algumas ISOs e frameworks como Scrum e XP, não havia algo consolidado que orientasse desenvolvedores especificamente na qualidade do código-fonte. Foi então que surgiu o **Código Limpo**.

O termo "Código Limpo" abordado nesta obra vem da expressão em inglês ***Clean Code***, que é um conjunto de práticas de programação descritas no livro de mesmo nome escrito por *Robert Cecil Martin*, em 2008.

As práticas de Código Limpo não são normas nem convenções. São orientações que visam ajudar os programadores a criarem códigos mais coerentes, organizados, fáceis de ler e, consequentemente, de dar manutenção.

QUEM É ROBERT CECIL MARTIN?

Robert C. Martin, também conhecido como *Uncle Bob* (tio Bob, em português), é uma referência mundial no desenvolvimento de software. Atua nessa área desde 1970 e já escreveu vários livros:

- *Agile Software Development, Principles, Patterns, and Practices* (2002);
- *UML for Java Programmers* (2003);
- *Agile Principles, Patterns, and Practices in C#* (2006);
- *Clean Code* (2008);
- *The Clean Coder* (2011);
- *Clean Architecture* (2017);
- *Clean Agile* (2019).

Em 2000, escreveu sobre um conjunto de princípios hoje chamados de **SOLID**, sobre os quais há um pequeno resumo no capítulo sobre *Classes*. Esses princípios são considerados uma referência na avaliação de qualidade de código. É também um dos autores do **Manifesto Ágil**, publicado em 2001. Esse documento pode ser considerado o documento inicial que inspirou a criação de tudo que adotamos como *Ágil* no desenvolvimento de software, como o *Scrum* e o *XP*.

As questões abordadas nessas práticas são:

- Nomes de entidades;
- Tamanho de funções;
- Comentários de código;
- Formatação de código;
- Distinção entre objetos "inteligentes" e estruturas de dados simples;
- Retorno de `null` em funções;
- Limites entre entidades;
- Testes unitários;
- Coesão de classes;
- Injeção de dependências / programação orientada a aspectos;
- Testes automatizados e Refatoração;
- Programação concorrente.

Embora essas práticas tenham sido elaboradas para serem aplicadas em Programação Orientada a Objetos (POO), algumas delas podem ser usadas em outros paradigmas de programação.

1.1 O importante é que funcione?

Já trabalhei com pessoas que diziam que, na programação de software, ***o importante é que funcione***. Será que é isso mesmo? Será que só o resultado importa? Vamos refletir sobre a seguinte situação: *João Kacau é sobrevivente de um naufrágio. Conseguiu ir do barco afundando até a margem, que ficava a cerca de 50 metros do acidente.*

João fez os movimentos suficientes para se mover 50 metros pela água até a margem e se salvar. João sabe nadar? Ele nadou de forma correta? Nesse caso, podemos dizer que **João se salvou!**

Isso é o que importa! Não importa COMO ele fez para chegar à margem!

Fique com esse episódio do naufrágio em mente. Vamos refletir agora sobre outra situação: *João Kacau foi contratado para desenvolver um programa para um pequeno consultório, que não trabalha com planos de saúde e só recebe pagamento em dinheiro. O prazo era de 2 meses.*

João pensou: *Vou programar rápido e de forma que o programa fique como o cliente espera. O programa é pequeno mesmo e o importante é que funcione!* Começou então o desenvolvimento... Após duas semanas de programação, João teve que voltar a mexer com códigos que fez no início do trabalho. Mas, como era um código desorganizado em uns lugares e sem padrão em outros, levou muito tempo para mexer no código que ele mesmo criou.

Após um mês de programação, João achou que precisava de outro desenvolvedor para cumprir o prazo. O novo colega se deparou com códigos desorganizados e/ou sem padrão. Assim, João tinha que, muitas vezes, parar o que estava fazendo para explicar seu código para o novo colega. O novato não estava muito satisfeito com o código legado com o qual tinha que trabalhar, logo, sua motivação não era das melhores.

João dizia ao novato: *"Codifique o mais rápido que puder. Só garanta que tudo funcione certo, pois só me importa o resultado!"*. E o novato foi criando mais e mais código de qualidade duvidosa no projeto.

Após três meses (ou seja, com um mês de atraso), João e seu parceiro entregam o programa. De fato, ele até fazia tudo o que fora solicitado. Nesse instante, João comemorou: *O importante é que funcionou!*

Porém, após algum tempo de uso foi encontrado um pequeno bug. Mesmo sendo pequeno, João e o parceiro demoraram uma semana para achar a causa e eliminá-lo, devido à baixa qualidade no código.

Detalhe: eles cobraram o equivalente a quatro horas de trabalho pois foi o tempo que acharam que levariam para corrigir o bug.

Após um ano de uso, o cliente chamou João e fez pedidos de novas funcionalidades para o programa.

- Gostaria de poder atender a planos de saúde;
- Gostaria de aceitar pagamento com cartão.

Como o código estava com baixa qualidade, desorganizado e sem padrão, levaram quase o dobro do tempo prometido só para implementar a primeira funcionalidade. Tentaram renegociar prazo e custo com o cliente, que cancelou o contrato.

Esse duro processo de desenvolvimento e triste fim do contrato aconteceram porque João Kacau achou que programar era como nadar para sobreviver a um naufrágio. **Mas não é!** Nadar para sobreviver é uma ação que não precisará ser consertada ou melhorada, muito menos ampliada. O desenvolvimento de um programa, definitivamente, não é assim!

Não estou dizendo aqui que o resultado final não importa, mas **como** se chega a ele é tão importante quanto. Além de ser mais difícil chegar ao programa funcionando como deveria sem dar valor a **como** se programou, trabalhar em equipe, realizar manutenções e criar novas funcionalidades serão tarefas mais e mais complexas.

1.2 As orientações de Código Limpo são usadas no mundo real?

Vários alunos já me perguntaram se o Código Limpo realmente é adotado no mercado de trabalho. Até pouco tempo eu respondia baseado em minhas experiências pessoais e no que eu podia observar em redes sociais, principalmente no Linkedin. Porém, um estudo científico de 2022 se propôs a, literalmente, perguntar aos

desenvolvedores se usavam o Código Limpo. E minha pesquisa de mestrado, realizada em 2024, também queria descobrir isso, mas com foco na indústria de TI brasileira.

Estudo sobre adoção de Código Limpo na Europa

Segundo o estudo realizado por Kevin Ljung e Javier Gonzalez-Huerta (2022) junto a profissionais da Europa, quase 90% dos profissionais de desenvolvimento participantes da pesquisa tinham conhecimento pelo menos superficial das técnicas de Código Limpo. Já 77% tinham um bom conhecimento sobre elas. É possível ver melhor essa informação na figura a seguir.

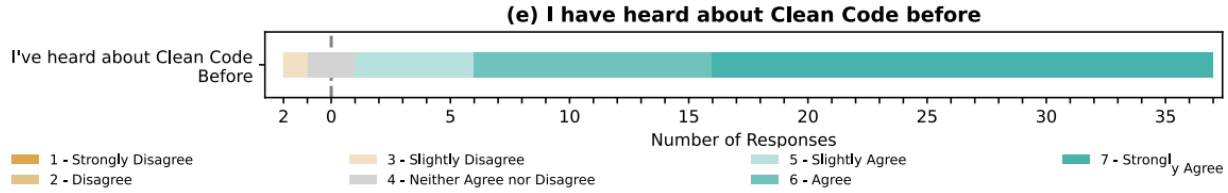


Figura 1.1: "Já ouviu falar de Código Limpo?". Fonte: LJUNG; GONZALEZ-HUERTA, 2022.

Na mesma pesquisa, quando perguntados se concordavam com os princípios gerais de Código Limpo, o princípio que teve a menor adesão teve aproximadamente 82% de apoio. Veja na próxima figura.

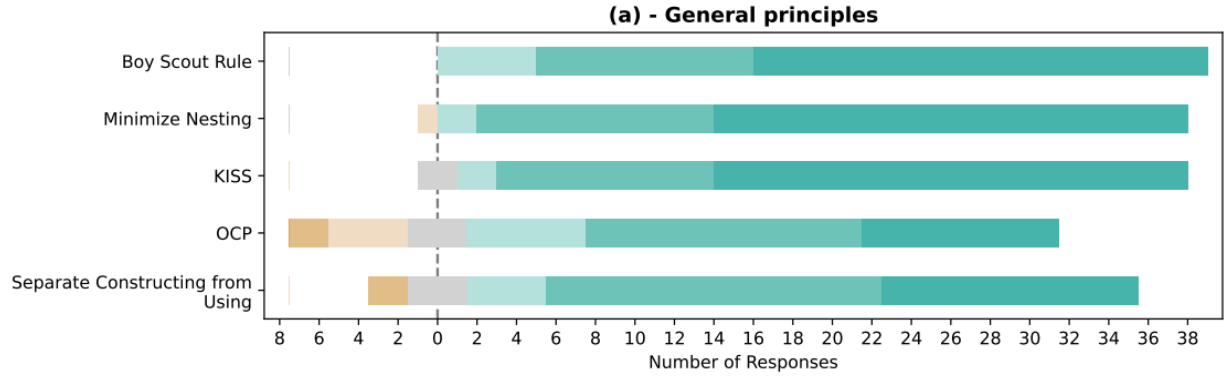


Figura 1.2: "Você concorda com os princípios gerais de Código Limpo?". Fonte: LJUNG; GONZALEZ-HUERTA, 2022.

Dentre os princípios de Código Limpo, o que teve menor adesão pelos entrevistados foi sobre uma asserção por teste (assunto que é abordado no capítulo 8). Veja na figura a seguir.

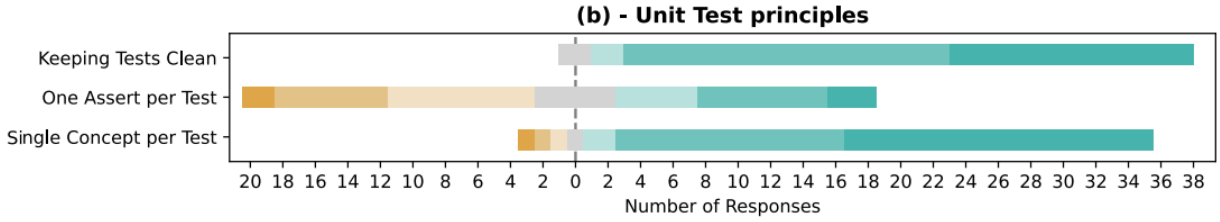


Figura 1.3: "Você concorda com os princípios sobre teste de unidade do Código Limpo?". Fonte: LJUNG; GONZALEZ-HUERTA, 2022.

Já quando perguntados sobre os benefícios de lidar com um código limpo no dia a dia, as respostas foram muito positivas, como é possível ver na próxima figura.

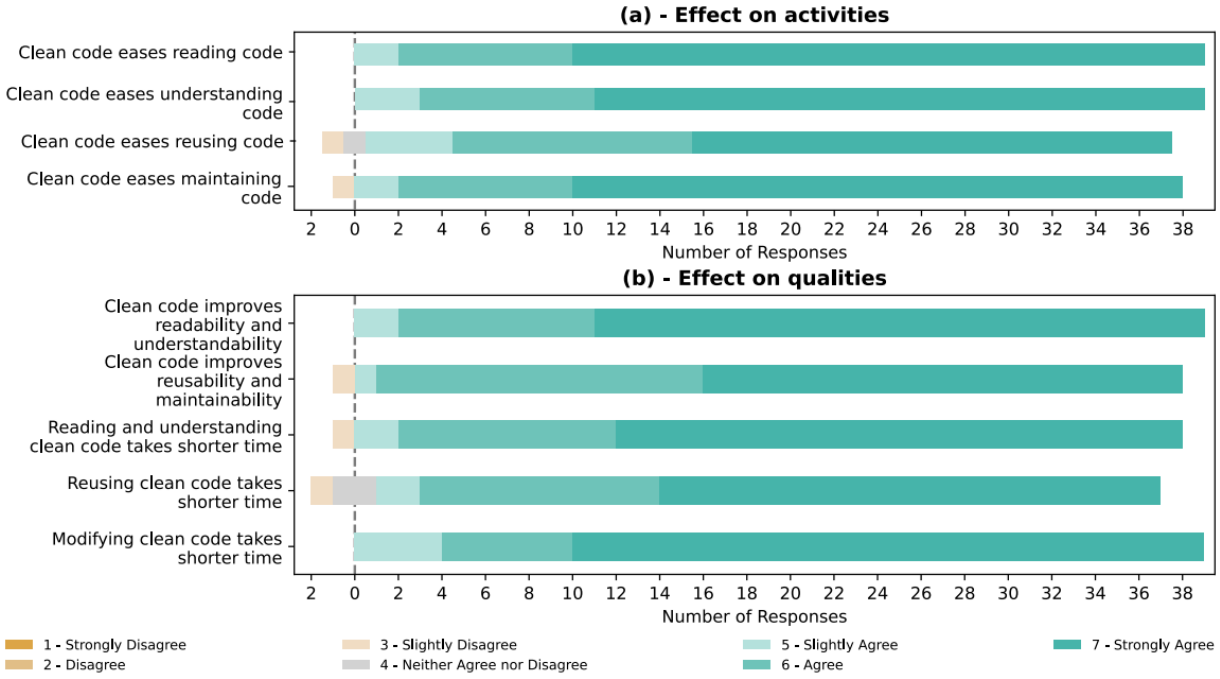


Figura 1.4: Percepções dos desenvolvedores sobre se um código limpo (a) facilita as tarefas de leitura, compreensão, reutilização e manutenção do código; e (b) seu impacto na legibilidade, compreensão, manutenção e velocidade de interação. Fonte: LJUNG; GONZALEZ-HUERTA, 2022.

Estudo sobre adoção de Código Limpo no Brasil

A pesquisa do meu mestrado teve como um dos objetivos saber se as práticas de Códido Limpo eram realmente usadas no Brasil. O trabalho foi defendido e aprovado na USP em outubro de 2024. Nessa pequisa agrupei as práticas/princípios de Código Limpo em 15 (quinze). Em resumo, os resultados desse objetivo da pesquisa foram:

- Apenas **1** das práticas/princípios teve **menos de 50% de resposta positiva** em relação à adoção
- **6 das 15** práticas/princípios tiveram **80% ou mais de resposta positiva** em relação à adoção

Ou seja: segundo a pesquisa, o Código Limpo tem uma ótima adesão no Brasil.

Na figura a seguir é possível ver uma foto dessa parte da pesquisa.

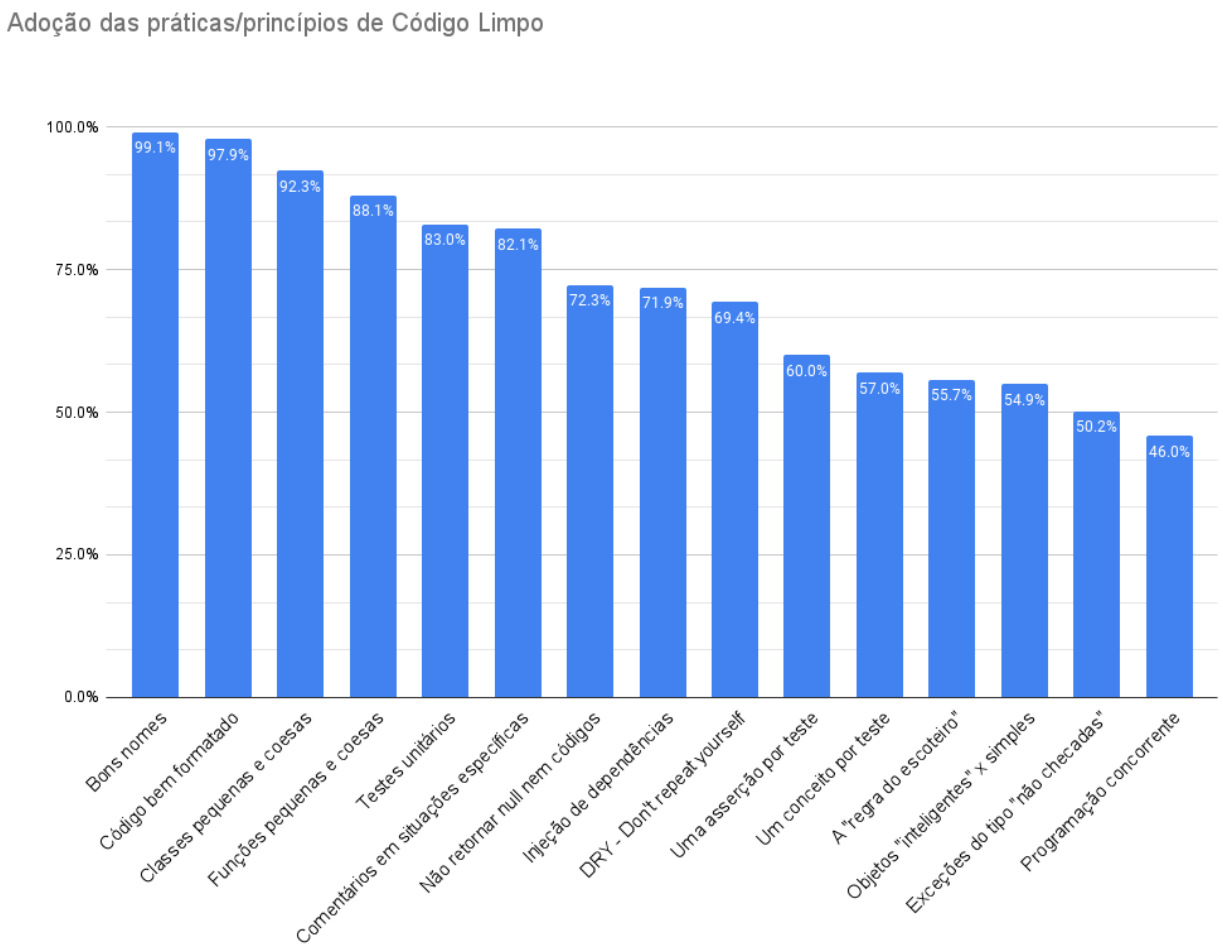


Figura 1.5: "Adoção das práticas/princípios de Código Limpo". Fonte: YOSHIRIRO, 2024.

Outro objetivo da pesquisa era saber se o Código Limpo era ensinado na educação formal de nível superior. Em resumo, os resultados indicam que o ensino nas faculdades ainda está abaixo do que o mercado pede. Dentre os resultados, destaco que **7 das 15** práticas/princípios tiveram **menos de 50% de resposta positiva** em relação ao ensino formal. E **6 das 15** práticas/princípios tiveram uma assimetria muito grande entre o quão são ensinadas versus o quão são adotadas no mercado. Em outras palavras: são pouco ensinadas nas faculdades, porém muito usadas pela indústria. É possível ver essa assimetria na figura a seguir.

Prática/Princípio	% dos que aprenderam na IES	% dos que adotam
Usar comentários de código apenas em situações muito específicas	44.7%	82.1%
Não retornar "null" ou "códigos de retorno" em métodos, optando por lançar exceções	47.2%	72.3%
Injeção de dependências e/ou programação orientada a aspectos	39.6%	71.9%
DRY - *Don't repeat yourself*	36.2%	69.4%
Um conceito por teste	31.9%	57.0%
A "regra do escoteiro"	24.7%	55.7%

Figura 1.6: "Práticas/princípios de Código Limpo com maiores assimetrias entre o ensino e adoção". Fonte: YOSHIRIRO, 2024.

1.3 O código deve ser 100% limpo?

Se você é iniciante no universo da programação, talvez seja uma boa ideia seguir as orientações de Código Limpo até ter a maturidade e experiência necessárias para saber quando não seguir uma ou outra orientação.

Nossa, quer dizer que o próprio autor do livro está dizendo que não devo seguir tudo que está nele?

Como já dito, Código Limpo não são normas. São orientações para diversas questões relacionadas diretamente à programação. Pense nesse guia como um conjunto de práticas de uma arte marcial. Você será orientado a dar chutes de uma certa forma. Porém, você conhece uma outra arte marcial na qual percebe que o chute que ela ensina é melhor para você. Sem problema, use o chute da outra arte marcial. O importante é chutar da maneira que mais lhe favoreça. As práticas de Código Limpo não são as únicas que deixam um código limpo.

Aqui você terá contato com as práticas de Código Limpo acompanhadas de exemplos em Java e Python. Se alguma delas não lhe convencer, sem problema: não a utilize, ou a adapte à sua realidade. Adote apenas as práticas que derem *match* com seu estilo e/ou necessidade.

Tão importante quanto as práticas em si são os objetivos que elas buscam para seu código-fonte:

- Organização;
- Facilidade de leitura e entendimento;
- Facilidade na localização de bugs;
- Facilidade na manutenção e evolução do projeto, ou seja, facilitar o uso do código-fonte já existente;
- Ajuda no trabalho em equipe.

Portanto, se for adaptar ou se negar a usar algumas das práticas de Código Limpo, faça-o em vista desses objetivos. Considere que boa parte do código que você escrever provavelmente será lida e/ou mexida por outra pessoa no futuro — ou ainda que você lide com seu próprio código no futuro, não seria melhor pegar um código limpo?

Por fim, a minha recomendação é que um código deve ser 100% limpo, mesmo que não siga 100% as orientações de Código Limpo cunhadas por Robert C. Martin.

1.4 Clean Code x Clean Arch (Código limpo x Arquitetura Limpa)

Existe uma pequena confusão, principalmente entre iniciantes em programação, em achar que Código Limpo e **Arquitetura Limpa** (do termo original em inglês *Clean Arch* ou *Clean Architecture*) são a

mesma coisa. Ambos os termos foram cunhados pelo Robert C. Martin (*Uncle Bob*), mas **não** são a mesma coisa!

Enquanto o Código Limpo tem como "ano de nascimento" 2008, com a publicação do livro mencionado no início deste capítulo, o termo **Arquitetura Limpa** foi mencionado por Robert Martin pela primeira vez em 2012, em seu blog. E o livro entitulado *"Clean Architecture: A Craftsman's Guide to Software Structure and Design"* (*"Arquitetura Limpa: o Guia do Artesão Para Estrutura e Design de Software"* na versão traduzida aqui no Brasil) foi publicado em 2017.

Código Limpo é um conjunto de princípios e práticas que têm como foco a qualidade do código-fonte.

Arquitetura Limpa é uma proposta de arquitetura cujo foco é a organização do projeto como um todo, definindo como as diferentes partes do software se relacionam e como o sistema pode evoluir com o tempo. A ideia é organizar o código em camadas e componentes para aumentar a facilidade de manutenção, facilitar a implementação de testes, permitir mudanças na tecnologia (bibliotecas e frameworks) sem afetar todo o sistema e ainda promover a modularidade. O diagrama clássico que ilustra essa arquitetura está na figura a seguir.

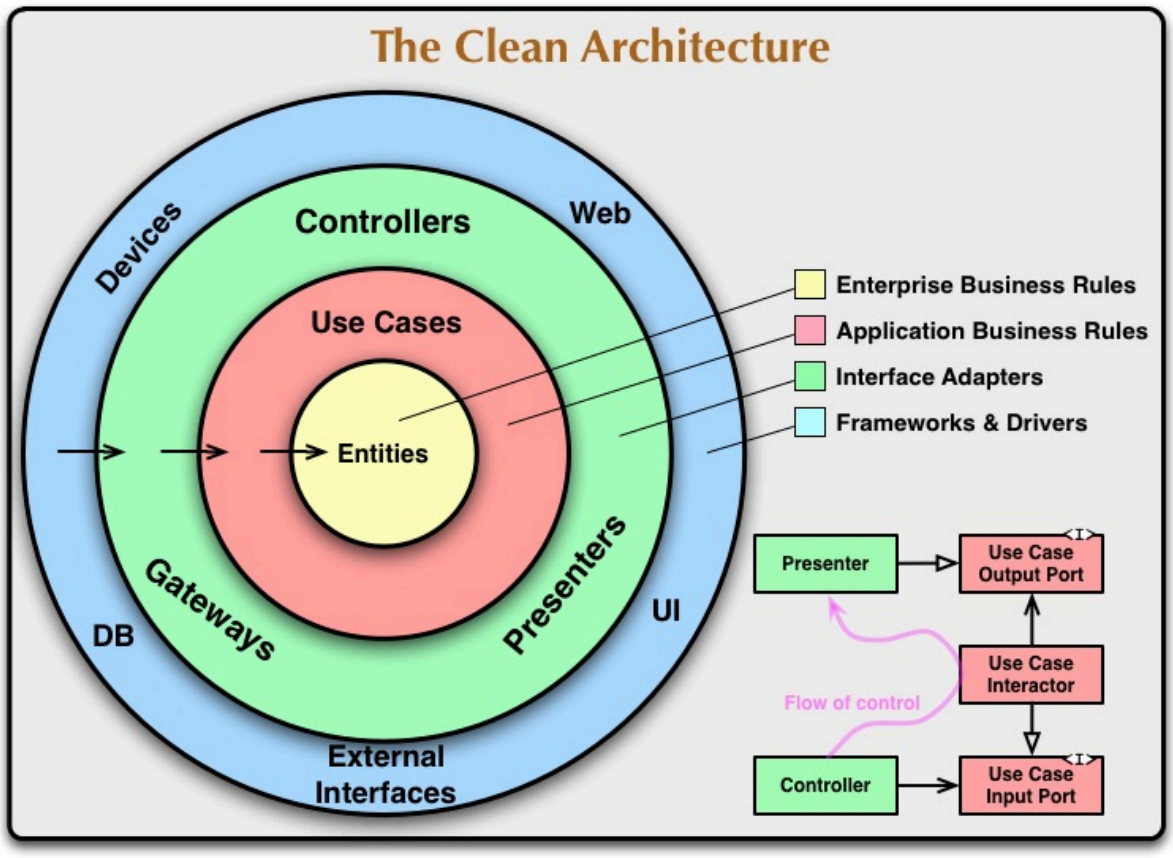


Figura 1.7: "Camadas e componentes da Arquitetura Limpa". Fonte: MARTIN, 2012.

A seguir, uma tabela com o resumo das diferenças entre eles.

Característica	Código Limpo	Arquitetura Limpa
Nível	Código-fonte	Estrutura do projeto
Objetivo	Facilitar a leitura e manutenção do código	Criar sistemas robustos e flexíveis
Práticas	Nomes, funções, comentários, formatação, exceções, testes,	Separação de responsabilidades, camadas, inversão de dependência

Característica	Código Limpo	Arquitetura Limpa
	regra do escoteiro etc.	

Adotar Código Limpo obriga a adotar Arquitetura Limpa?

Absolutamente **não**. É possível usar o Código Limpo em um projeto mas usar outras arquiteturas, como a Arquitetura Hexagonal, por exemplo. Assim como em um projeto no qual foi usada a Arquitetura Limpa não há garantia alguma de que seu código-fonte segue o Código Limpo.

Capítulo 2
Bons nomes

João Kacau chega a uma cozinha com a tarefa de preparar um determinado prato. Ao chegar lá, encontra diversos potes de mesmo tamanho e formato, todos contendo um pó branco e nenhum com identificação. Então, usando a visão, tenta descobrir o que é sal, açúcar refinado, trigo ou outro ingrediente, mas ainda assim surgem dúvidas. Tenta o olfato. Ainda não há 100% de certeza. Tenta o paladar. Finalmente consegue saber o que há em cada pote. Um pequeno detalhe: João detesta pimenta (ele acha até refrigerante forte) e descobre, com o paladar, que um dos potes tem um tipo de pimenta bem picante.

O que você diria sobre a pessoa que deixou a cozinha desse jeito? Gostaria de cozinhar em uma cozinha preparada por ela? Eis que João lembrou-se de que foi ele mesmo quem preparou a cozinha há alguns dias. Ele havia pensado *"qualquer um vai saber o que há nos potes, basta olhar de perto o conteúdo!"* e, ainda, *"se eu mesmo for cozinhar depois, claro que vou me lembrar de como deixei as coisas"*.

Fazemos algo semelhante quando não damos bons nomes às entidades em nossos programas: criamos um problema futuro, que é o de descobrir para que elas servem. Isso pode ser chato, demorado e até perigoso!

Criar variáveis ou funções com nomes como `x`, `y`, `a` e `b` certamente nos fazem escrever menos, mas dificilmente deixará nosso código com mais qualidade.

Opinião profissional

"Os nomes devem ser claros e representar exclusivamente o propósito dos elementos. Além disso, deve haver um padrão de nomenclatura para toda a base de código — bem como uma linguagem. Aborrece encontrar no mesmo software as funções 'printOutput', 'printResult' e 'imprimir'."

Renan Rodrigo, mestre em Ciência da Computação pelo IME-USP.

A seguir, vejamos as técnicas de Código Limpo para dar nomes às entidades.

2.1 Use nomes que deixem claro o objetivo da entidade

Imagine que você está criando um programa que implementa um cofrinho, aquele simpático objeto onde depositamos dinheiro. Nesse programa você vai precisar lidar com o *nome do dono*, *valor atual* no cofre e a *quantidade de depósitos realizados*. Que nomes você daria para as variáveis que armazenariam esses dados? Vejamos os exemplos a seguir.

Java:

```
String nom;
Double atual;
Integer deps;
```

Python:

```
nom
atual
deps
```

Essas variáveis têm nomes curtos, certo? Mas será que estão bons? Vejamos:

- `nom` : Seria "nome"? Mas nome do quê? Do cofrinho? Do objetivo do cofrinho? Do dono?
- `atual` : Atual é melhor que defasado, hein? Brincadeiras à parte, quem criou a variável talvez lembre o motivo de ter dado esse nome, isso se não faz mais de uma semana que esse código foi escrito.
- `deps` : Depois? Dependentes? Departamentos?

Ora, se o objetivo era lidar com o nome do dono, o valor atual no cofre e a quantidade de depósitos realizados, nomes melhores poderiam ser como nos exemplos a seguir.

Java:

```
String nomeDono;
Double valorAtual;
Integer depositos;
```

Python:

```
nome_dono
valor_atual
depositos
```

Ou, se formos um pouco mais técnicos no que diz respeito a investimentos financeiros, sabemos que o valor total em um investimento costuma ser chamado de `saldo` .

Java:

```
String nomeDono;
Double saldo;
Integer depositos;
```

Python:

```
nome_dono
saldo
depositos
```

Percebeu como os nomes agora deixam bem mais claro quais informações serão armazenadas nas variáveis? Você pensou em outros nomes claros e bons para elas?

Precisamos também de uma função que permita realizar um *depósito* no cofrinho. Vejamos como ela está nos próximos exemplos.

Java:

```
public void dep(Double v) {
    // corpo da função
}
```

Python:

```
def dep(v):
    # corpo da função
```

Em ambos os exemplos fizemos uma boa economia de letras, mas será que os nomes da função e de seu parâmetro estão bons? O que é `dep` ? Esse nome deixa claro que o objetivo da função é receber um depósito no cofrinho? O que é `v` ? Esse nome deixa claro que se trata do valor a ser depositado no cofre?

Se o objetivo da função é realizar um depósito a partir do valor recebido no parâmetro, os nomes ficariam melhores como nos próximos exemplos.

Java:

```
public void depositar(Double valorDeposito) {
    // corpo da função
}
```

Python:

```
def depositar(valor_deposito):
    # corpo da função
```

E agora? Resta alguma dúvida sobre o objetivo do método? E sobre o objetivo do parâmetro? Você teria outras sugestões de nomes para a função e seu parâmetro?

E se estivermos programando orientado a objetos e criamos uma classe para representar o conceito de cofrinho? O que acha desse nome de classe?

Java:

```
public class CalcCofre {
    // corpo da classe
}
```

Python:

```
class CalcCofre():
    # corpo da classe
```

Apesar de o termo "Cofre" no nome dar uma dica, o nome `CalcCofre` não pode ser considerado claro. Que tal simplesmente `Cofrinho` ? Dessa forma, a classe ficaria como no código a seguir.

Java:

```
public class Cofrinho {
    // corpo da classe
}
```

Python:

```
class Cofrinho():
    # corpo da classe
```

Note que, quando instanciarmos um objeto dessa classe, teremos criado um *cofrinho*, assim como indica, claramente, o novo nome.

2.2 O nome não deve causar confusão sobre o que a entidade é

Em um dos exemplos anteriores sugerimos que a variável que armazena a quantidade de depósitos já realizados seja chamada de `depositos`. Sem dúvida ficou melhor que o nome anterior (`deps`). Porém, ainda assim pode causar uma certa confusão. Note que `depositos` parece também ser nome de vetor ou lista, não concorda? Assim, que tal melhorarmos ainda mais o nome dessa variável? Vejamos a nova proposta de nome a seguir.

Java:

```
Integer quantidadeDepositos;
```

Python:

```
quantidade_depositos
```

Com esse novo nome, fica bem difícil de achar que essa variável é um vetor ou uma lista.

Vejamos outros exemplos de nomes quase bons, mas que podem gerar dúvidas.

Java:

```
Integer mediaDepositos1a3;
```

Python:

```
media_depositos_1_a_3
```

Trata-se da média dos três primeiros depósitos ou dos depósitos de valor entre 1,00 e 3,00? Bom, se for a primeira opção, que tal os nomes a seguir?

Java:

```
Integer media3PrimeirosDepositos;
```

Python:

```
media_3_primeiros_depositos
```

2.3 Cuidado com caracteres parecidos — 0, O, 1, l

Tenha cuidado com caracteres parecidos entre si, pois podem ser confundidos com outros. Vamos supor que nosso programa de cofrinho tem um requisito (que revelarei depois) que nos levou a criar a seguinte variável.

Java:

```
double deposito01;
```

Python:

```
deposito_01
```

Dependendo da fonte usada, talvez você não tenha certeza se o penúltimo caractere é um zero ou uma letra "o" maiúscula. Uma outra dúvida que pode surgir é se o último caractere é um número um ou uma letra "l" minúscula.

A propósito, essa variável foi criada para atender o requisito de sempre poder ser possível saber o valor do primeiro depósito no cofrinho. Um nome melhor para ela poderia ser como no próximo exemplo.

Java:

```
double primeiroDeposito;
```

Python:

```
primeiro_deposito
```

Atenção! É claro que você pode usar esses quatro caracteres — 0 (zero), O (letra "o" maiúscula), 1 (um) e l (letra "l" minúscula) — para dar nomes. A orientação é observar se não ficaram dúvidas sobre qual caractere é usado no nome da entidade. Seguem exemplos de código onde não haveria esse tipo de confusão.

Java:

```
double mediaDepositos1a7 = 10.25;
String principalObjetivo = "Comprar um PS5";
```

Python:

```
media_depositos_1_a_7 = 10.25
principal_objetivo = 'Comprar um PS5'
```

2.4 Evite nomes parecidos

Um programa de verdade não vive só de uma variável, uma função ou uma classe. Por isso, pode haver confusão sobre o que uma entidade é quando temos nomes parecidos. Vamos supor que precisemos de mais três informações em nosso cofrinho: meta de curto prazo, meta de médio prazo e meta de longo prazo. Todas seriam apenas descrições sobre três possíveis metas para o dinheiro guardado. As variáveis que armazenariam esses dados poderiam ser como nos exemplos a seguir.

Java:

```
String metaDinheiroGuardadoCofrinhoCurtoPrazo;
String metaDinheiroGuardadoCofrinhoMedioPrazo;
String metaDinheiroGuardadoCofrinhoLongoPrazo;
```

Python:

```
meta_dinheiro_guardado_cofrinho_curto_prazo
meta_dinheiro_guardado_cofrinho_medio_prazo
```

```
meta_dinheiro_guardado_cofrinho_longo_prazo
```

Se escolhermos uma dessas três variáveis para analisar, parece que cada uma delas tem um ótimo nome, que revela bem a intenção da variável. Porém, ao olharmos todas essas linhas juntas, ou se imaginarmos como uma IDE sugeriria uma delas naquele recurso de *autocomplete* de código (observe na próxima figura), percebemos que existe uma boa possibilidade de usarmos a variável errada.

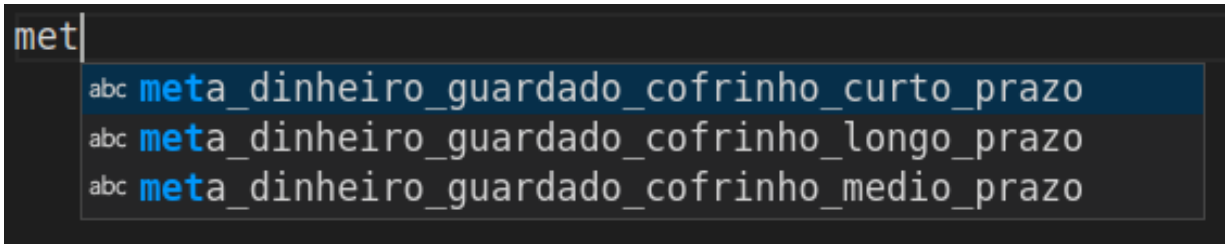


Figura 2.1: Autocomplete de código com variáveis de nome parecido.

Os nomes das variáveis poderiam ser melhorados para evitar confusão, como nos próximos exemplos.

Java:

```
String metaCurtoPrazo;
String metaMedioPrazo;
String metaLongoPrazo;
```

Python:

```
meta_curto_prazo
meta_medio_prazo
meta_longo_prazo
```

Nesse caso, reduzir os nomes não tornou obscuro o objetivo das variáveis e reduziu bastante as chances de confusão entre elas.

Vejamos agora exemplos de nomes de funções que, isoladamente, até têm bons nomes, mas, como estão no mesmo local, passam a ter nomes parecidos, o que pode causar confusão.

Java:

```java
void definirNomeDonoCofrinho(String nome) {
    // corpo da função
}

void definirNomeCofrinho(String nome) {
    // corpo da função
}

void definirNomeObjetivoCofrinho(String nome) {
    // corpo da função
}
```

Python:

```python
def definir_nome_dono_cofrinho(nome):
    # corpo da função

def definir_nome_cofrinho(nome):
    # corpo da função

def definir_nome_objetivo_cofrinho(nome):
    # corpo da função
```

Note a similaridade do nome do método: todos começam com `definir`, contêm `nome` e terminam com `cofrinho`. Imagine esses três nomes sendo sugeridos por um assistente de *autocomplete* de IDE.

Os nomes dos parâmetros são todos iguais (`nome`). Como as IDEs costumam mostrar o nome do parâmetro nas sugestões de código, isso realmente pode favorecer o uso da função errada durante a codificação.

Vamos, portanto, deixar os nomes das funções e de seus parâmetros melhores como nos códigos a seguir.

Java:

```java
void definirDono(String nomeDono) {
    // corpo da função
}
```

```
void definirNomeCofrinho(String nomeCofrinho) {
    // corpo da função
}

void definirObjetivo(String objetivoCofrinho) {
    // corpo da função
}
```

Python:

```
def definir_dono(nome_dono):
    # corpo da função

def definir_nome_cofrinho(nome_cofrinho):
    # corpo da função

def definir_objetivo(objetivo_cofrinho):
    # corpo da função
```

Java:

```
void definirDono(String nomeDono) {
    // corpo da função
}

void definirNomeCofrinho(String nomeCofrinho) {
    // corpo da função
}

void definirObjetivo(String objetivoCofrinho) {
    // corpo da função
}
```

Python:

```
def definir_dono(nome_dono):
    # corpo da função

def definir_nome_cofrinho(nome_cofrinho):
    # corpo da função
```

```python
def definir_objetivo(objetivo_cofrinho):
    # corpo da função
```

2.5 Ações semelhantes, nomes semelhantes

Imagine que o programa que simula um cofrinho tem as seguintes funções:

Java:

```java
String retornaDono() {
    return this.dono;
}

String obterNomeCofrinho() {
    return this.nomeCofrinho;
}

String objetivo() {
    return this.objetivoCofrinho;
}
```

Python:

```python
def retorna_dono(self):
    return self._dono

def obter_nome_cofrinho(self):
    return self._nome_cofrinho

def objetivo(self):
    return self._objetivo_cofrinho
```

Todas essas funções têm um objetivo parecido: retornar o valor de um atributo. Ao olharmos individualmente, cada nome até que está

bom. Porém, mesmo tendo objetivos parecidos, temos nomes sem um padrão definido. Isso pode causar questionamentos do tipo:

- Por que um é `retorna...` e o outro é `obter...` ? Será que isso causa alguma diferença de comportamento?
- Se eu for criar uma função que retorna o valor de um atributo, que prefixo uso? Ou não uso, como na `objetivo()` ?

Para evitar dúvidas desnecessárias como essas, siga ou defina um padrão em entidades que possuem ações semelhantes. Por exemplo, nossas funções poderiam ficar como nos códigos a seguir.

Java:

```
String obterDono() {
    return this.dono;
}

String obterNomeCofrinho() {
    return this.nomeCofrinho;
}

String obterObjetivo() {
    return this.objetivoCofrinho;
}
```

Python:

```
def obter_dono(self):
    return self._dono

def obter_nome_cofrinho(self):
    return self._nome_cofrinho

def obter_objetivo(self):
    return self._objetivo_cofrinho
```

Os novos nomes das três funções são agora tão parecidos quantos as ações que elas fazem. Isso facilita a vida de quem for apenas invocá-las e de quem for criar novas funções com finalidades parecidas, pois não precisará pensar muito no nome, já que existe um padrão.

Esse tipo de problema poderia ocorrer com nomes de variáveis e de classes também. Vamos imaginar que nosso programa de cofrinho precise de classes utilitárias para validar as entradas financeiras e de texto do programa, como nos exemplos a seguir.

Java:

```
public class ValidadorNumeros {
    // corpo da classe
}

public class TextosUtil {
    // corpo da classe
}
```

Python:

```
class ValidadorNumeros():
    # corpo da classe

class TextosUtil():
    # corpo da classe
```

Ambas as classes têm um objetivo parecido: validar valores (uma valida números e a outra, textos). Ao olharmos individualmente, cada nome até que está bom. Porém, mesmo tendo objetivos parecidos, temos nomes sem um padrão definido. Isso pode causar questionamentos do tipo:

- Por que um é `Validador...` e o outro é `...Util` ? Será que isso causa alguma diferença de comportamento?
- Se eu for criar uma classe de validação de datas, chamo de `ValidadorDatas` ou `DatasUtil` ?

Para evitar essas dúvidas, padronizar os nomes das classes seria a solução, como no código a seguir.

Java:

```java
public class ValidadorNumeros {
    // corpo da classe
}

public class ValidadorTextos {
    // corpo da classe
}
```

Python:

```python
class ValidadorNumeros():
    # corpo da classe

class ValidadorTextos():
    # corpo da classe
```

Com os novos nomes, não resta dúvidas de que todo `Validador...` tem uma ação semelhante e que os próximos devem ser criados com esse prefixo.

2.6 Mesma palavra, mesmo propósito

Em complemento às últimas recomendações, fica uma dica sobre ter cuidado com algo que costuma acontecer principalmente em projetos médios e grandes e/ou em projetos com muitos colaboradores. Observe os próximos códigos.

Java:

```
List simularLongoPrazo() {
    // código que cria uma simulação em uma lista e essa lista fica
disponível para consultas depois
}

void simularRendimento(double valorDepositoMensal) {
    // código que envia o resultado de uma simulação por e-mail
}

double simularJuros() {
    // código que retorna o cálculo de uma simulação de juros
}
```

Python:

```
def simular_longo_prazo() {
    # código que cria uma simulação em uma lista e essa lista fica
disponível para consultas depois
}

def simular_rendimento(valor_deposito_mensal) {
    # código que envia o resultado de uma simulação por e-mail
}

def simular_juros() {
    # código que retorna o cálculo de uma simulação de juros
}
```

Note que as três funções, apesar de realizarem uma simulação, possuem ações diferentes. A primeira preenche variável, a segunda envia um resultado por e-mail e a terceira devolve imediatamente o resultado de um cálculo. Assim, manter o prefixo `simular` seguido de uma ação dá o falso indicador de que terão comportamentos muito parecidos. Logo, para que os nomes dos métodos não causem esse tipo de mal-entendido, seus nomes ficariam melhores evitando o mesmo termo no início do nome, como nos próximos códigos.

Java:

```
List criarSimulacaoLongoPrazo() {
    // código que cria uma simulação nula lista e essa lista fica
disponível para consultas depois
}

void enviarEmailSimulacaoRendimento(double valorDepositoMensal) {
    // código que envia o resultado de uma simulação por e-mail
}

double obterSimulacaoJuros() {
    // código que retorna o cálculo de uma simulação de juros
}
```

Python:

```
def criar_simulacao_longo_prazo() {
    # código que cria uma simulação nula lista e essa lista fica
disponível para consultas depois
}

def enviar_email_simulacao_rendimento(valor_deposito_mensal) {
    # código que envia o resultado de uma simulação por e-mail
}

def obter_simulacao_juros() {
    # código que retorna o cálculo de uma simulação de juros
}
```

2.7 Se travou a língua ou causou constrangimento, mude o nome

Travar a língua pode ser engraçado em situações como conversas durante o intervalo do cafezinho, em um *happy hour* ou em vídeos como o de um famoso apresentador brasileiro pedindo para uma professora de língua portuguesa falar a palavra fictícia "Papibaquígrafo" (e se dando mal ao final). Será que em um código

limpo existe espaço para palavras que travam a língua? Imagine que você se depara com a seguinte variável no código do cofrinho:

Java:

```
Double totalDepositadoHebdomadario = // um cálculo qualquer;
```

Python:

```
total_depositado_hebdomadario = # um cálculo qualquer
```

Sem dar aquela *googlada*, saberia dizer o objetivo dessa palavra? Como não? Vai dizer que não sabe que "hebdomadário" significa "semanal"? Bom, sinceramente, eu só descobri ao escrever este capítulo. Friamente falando, a variável tem um bom nome, afinal armazena o total depositado hebdomadário, ou seja, semanal. Mas o termo "semanal" é infinitamente mais conhecido que "hebdomadário", não concorda? O que justificaria usar uma palavra trava-língua e rara ao invés de uma de fácil leitura e muito, muito, mas muito mais conhecida?

Assim, a variável poderia simplesmente ser:

Java:

```
Double totalDepositadoSemanal = // um cálculo qualquer;
```

Python:

```
total_depositado_semanal = # um cálculo qualquer
```

Após navegar mais no código do cofrinho, achamos a seguinte função:

Java:

```
Boolean existeCornucopiaDeDepositos() {
    // retorna true ou false de acordo com alguma regra
}
```

Python:

```
def existe_cornucopia_de_depositos():
    # retorna true ou false de acordo com alguma regra
```

Em algumas regiões do Brasil, alguns poderiam dizer que saber o que a função retorna é "serviço de corno". Mas a função, afinal, diz claramente sua função. Ora, "cornucópia" significa "abundância"; portanto, a função retorna `true` se houve uma certa quantidade de depósitos considerada abundante ou `false` em caso contrário. Além de a palavra ser peculiar e um tanto trava-língua, é um termo que pode ser um tanto constrangedor. A função ficaria bem melhor como nos exemplos a seguir.

Java:

```
Boolean existeAbundanciaDeDepositos() {
    // retorna true ou false de acordo com alguma regra
}
```

Python:

```
def existe_abundancia_de_depositos():
    # retorna true ou false de acordo com alguma regra
```

De repente, você se depara com uma classe que auxilia na validação de termos usados nas entradas, para evitar palavras ou frases impróprias. A classe é a seguinte:

Java:

```
public class ListaNegra {
    // corpo da classe
}
```

Python:

```
class ListaNegra():
    # corpo da classe
```

Se você não notou nada de estranho no nome é porque não anda acompanhando um recente movimento que vem desestimulando o uso de termos como "*slave*", "escravo", "*blacklist*", "lista negra". É que são, de certa forma, referências à escravidão.

> **GIGANTES DO SOFTWARE SUGEREM TERMINOLOGIAS SEM REFERÊNCIAS À ESCRAVIDÃO**
>
> Em meados de 2020, ganhou força um movimento entre gigantes do mundo do software para trocar termos que fazem referência à escravidão. Esse movimento teve apoiadores individuais de peso, como **Linus Torvalds** (o criador do Linux e do Git) e grandes empresas como **Microsoft**, **Twitter**, **LinkedIn**, **Ansible**, **Splunk**, e **JP Morgan**. Como exemplos de consequências práticas desse movimento, temos:
>
> - A troca do termo `master` por `main` no **GitHub**;
> - A troca do termo de `master` por `main`, de `slave` por `replica`, de `blacklist` por `blocklist` e de `whitelist` por `allowlist` no **MySQL**.
>
> Para mais detalhes:
>
> - https://www.wired.com/story/tech-confronts-use-labels-master-slave/
> - https://www.fudzilla.com/news/51167-torvalds-signs-off-on-banning-a-blacklist-of-racist-words

Portanto, para evitar constrangimentos com o termo "ListaNegra", essa classe poderia se chamar:

Java:

```java
public class TermosNaoPermitidos {
    // corpo da classe
}
```

Python:

```
class TermosNaoPermitidos():
    # corpo da classe
```

2.8 Evite caracteres especiais

Quando eu comecei a programar, em 1994, perguntei a um instrutor o motivo de não podermos usar caracteres especiais nos nomes de variáveis e funções. Quando falo "caracteres especiais", refiro-me a caracteres com acentos, *emojis* e caracteres de outros idiomas (japonês, russo, árabe etc.). Ele disse simplesmente que o código não seria compilado. Atualmente, será que linguagens como Java e Python aceitam qualquer caractere para dar nomes às suas entidades? Observe o código a seguir.

Java:

```
public class CálculoRendiment😄 {
    // corpo da classe
}
```

Python:

```
class CálculoRendiment😄():
    # corpo da classe
```

Foque no último caractere do nome da classe. Dependendo do tipo de livro que estiver lendo (físico, PDF ou algum e-book) você pode estar vendo um *emoji* de sorriso ou simplesmente um caractere ilegível, como um retângulo ou algo do tipo. Fato é que nenhum desses códigos será compilado justamente por causa desse último caractere. Demais linguagens modernas como JavaScript e Kotlin também não aceitariam esse tipo de caractere. Portanto, usar *emojis* poderia até ser criativo, mas vai além de não deixar o código limpo: torna o código impossível de usar. Ok, bastaria trocar esse "sorriso"

por um caractere comum, certo? Nesse caso, nosso código ficaria como o a seguir.

Java:

```java
public class CálculoRendimento {
    // corpo da classe
}
```

Python:

```python
class CálculoRendimento():
    # corpo da classe
```

Esse código certamente vai compilar em todas as linguagens de programação citadas há pouco. Mas será que está tudo certo nele? Observe o segundo caractere do nome da classe: é uma letra "a" com acento agudo. Esse tipo de caractere deve ser evitado, mesmo que não impeça a compilação do código.

Talvez, dependendo do seu tipo de livro, você sequer esteja conseguindo ler o segundo caractere do nome da classe. E, dependendo das configurações do editor e/ou do sistema operacional de outra pessoa da equipe, ela também não consiga ler esse caractere. Isso porque existem várias codificações de caracteres, como `UTF-8`, `ISO-8859-1` etc. Outro computador que pode sentir bastante o uso de caracteres especiais é uma máquina onde é feito o *deploy* de um projeto (uma máquina com *Jenkins*, por exemplo). Usar caracteres simples, além de ser mais rápido (já que a maioria dos caracteres especiais exigem dois ou três toques no teclado), evita problemas com codificações de caracteres.

Mais uma sutil vantagem em usar apenas caracteres simples: maior facilidade na busca por entidades, seja por assistentes em IDE, seja por ferramentas como o `grep` do Linux. Por isso, não teria problema o termo "Cálculo" estar escrito errado, ou seja, sem acento agudo. O código ficaria como o a seguir.

Java:

```java
public class CalculoRendimento {
    // corpo da classe
}
```

Python:

```python
class CalculoRendimento():
    # corpo da classe
```

2.9 Não precisa dizer o que a entidade é no nome

João Kacau usava um cofrinho para guardar seu dinheiro. Mas o cofrinho já não mais é suficiente para o dinheiro que ele ganha, pois subiu de cargo, passando a ganhar mais. Ele decide ir ao banco, onde ocorre o seguinte diálogo:

João: *"Bom dia, segurança Roberto. Posso entrar?"*

Roberto, o segurança: *"Olá, cliente João. Bom dia. Pode entrar, claro."*

João, ao chegar ao caixa: *"Oi, caixa Luzia. Gostaria de abrir uma conta neste banco."*

Luzia, a caixa: *"Então, cliente João, esse tipo de operação é com a gerente Marise."*

João, após chegar junto à gerente: *"Olá, gerente Marise. Gostaria de abrir uma conta, por favor."*

Marise, a gerente: *"Pois não, cliente João. Preencha essa ficha aqui, por gentileza."*

Marise, a gerente, após enviar uma impressão: *"Por favor, estagiário Edu, poderia pegar minha impressão lá na impressora?"*

Achou algo de estranho nesse diálogo? No cotidiano, nós não ficamos tratando as pessoas citando suas profissões junto ao nome. Então por que criaríamos uma classe com atributos e funções como a do código a seguir?

Java:

```
public class ClasseCofrinho {

    Double atributoValorAtual = 0;

    public void funcaoDepositar(Double parametroValorDeposito) {
        this.atributoValorAtual += parametroValorDeposito;
    }
}
```

Python:

```
class ClasseCofrinho():

    def __init__(self):
        self._atributo_valor_atual = 0

    def funcao_depositar(self, parametro_valor_deposito):
        self._atributo_valor_atual += parametro_valor_deposito
```

Além de dificultar buscas em projetos médios ou grandes, usar o tipo da entidade no próprio nome não indica melhor o objetivo da entidade e, sim, sua natureza. Ora, o que vem após o termo `class` só pode ser uma classe. O que está definido como atributo só pode ser um... atributo. O que está programado como função é... uma função. Uma variável na assinatura de uma função só pode ser um parâmetro. Note que mesmo sem dizer o que a entidade é, os nomes podem ser ótimos, como no exemplo a seguir.

Java:

```
public class Cofrinho {

    Double valorAtual = 0;

    public void depositar(Double valorDeposito) {
        this.valorAtual += valorDeposito;
    }
}
```

Python:

```
class Cofrinho():

    def __init__(self):
        self._valor_atual = 0

    def depositar(self, valor_deposito):
        self._valor_atual += valor_deposito
```

Quando o tipo da entidade pode ajudar no nome

Imagine que você está programando uma interface gráfica para seu sistema de cofrinho. Nele, há aquele componente no qual um valor alfanumérico pode ser digitado para ser usado como o nome do dono. Há também aquele componente conhecido como *lista suspensa* ou *combo* no qual constam algumas opções para o objetivo do cofrinho.

Que nome você daria para os objetos desses componentes? Talvez `nome` e `objetivo`? Não seria de todo ruim, mas esse tipo de componente gráfico costuma ter vários atributos e funções específicos.

Por exemplo, uma *lista suspensa* ou *combo*, da tecnologia Java Swing, é representada pela classe `JComboBox` e, para obter o valor selecionado, usamos a função `getSelectedItem()`. Ou seja, `objetivo` ainda não seria o objetivo de fato. Para obter isso, precisaríamos

fazer `objetivo.getSelectedItem()`. Daí esse valor você guardaria em uma variável com que nome? `objetivo`?

Um caso especial: Interfaces

No caso da linguagem de programação **C#**, é comum usarem a letra **I** no início dos nomes de *interfaces*. As orientações de Código Limpo não são nesse sentido. Porém, uma vez que isso é convenção em uma determinada linguagem, estando bem presente até mesmo em livros e documentação, nesse caso não seria um problema.

2.10 Evite mapeamentos mentais

Em enunciados de exercícios de algoritmos, é comum ver apenas uma letra (**X**, **Y**, **A**, **B** etc.) para se referir a valores informados pelo usuário ou resultados de cálculos feitos pelo programa. Por exemplo, um trecho de um exercício que pede dois números e exibe a média entre eles pode ser algo como: *Ao final exiba a frase "A média entre X e Y é Z", onde **X** e **Y** são os dois números informados pelo usuário e **Z** é a média aritmética entre eles*. Então uma pessoa iniciante em programação pode achar que não há problemas em criar variáveis `x`, `y` e `z` no código do algoritmo implementado.

Fazer esse tipo de coisa no código exige um mapeamento mental, pois o código-fonte não é suficiente para entender o objetivo das variáveis. Ou seja, ou quem programou leu e se lembra do enunciado para fazer o mapeamento entre as letrinhas do código e do enunciado; ou, pior: todo (ou quase todo) o código precisa ser lido para que se entenda o objetivo delas.

Assim, nada melhor do que criar um código limpo, para que alguém que esteja lendo o código pela primeira vez entenda o objetivo das variáveis, mesmo sem ter lido o enunciado da questão. Dessa forma, nomes como `numero1`, `numero2` e `media` ficariam bem melhores.

Mas e os famosos `i` ou `j` em loops? As letras `i` e `j` são quase onipresentes em livros e tutoriais de programação quando se ensina os loops (iterações) com `for`. Se o corpo desse loop for realmente pequeno, essas letras podem não ser problemáticas porque isso pode ser considerado até convenção.

Mas e o `it` de linguagens novas? Em linguagens como **Groovy** e **Kotlin**, estruturas de repetição costumam ter um nome automático para a variável de iteração, que é o `it`. Deixo a mesma recomendação sobre `i` e `j`: se o corpo desse loop for realmente pequeno, esse `it` pode não ser problemático, uma vez que é frequentemente citado nos livros e documentações dessas linguagens. Porém, se programar loop aninhados (um loop dentro de outro), aí sim é muito importante **não** usar o `it`, pois as chances de confusão no código serão bem grandes.

2.11 Nomes de classes

Neste capítulo já constam vários exemplos de nomes de classes. Notou algo de comum entre eles? Notou que nenhum indica uma *ação*? Todos são *substantivos* ou *adjetivos*, mas nenhum é um *verbo*.

Mas qual seria o problema em chamar uma classe de `Economizar` em vez de `Cofrinho`? O problema é que uma classe é composta de atributos (ou propriedades) e métodos (ou funções). Note que podemos dizer *"Um Cofrinho tem características e comportamentos, que são seus atributos e métodos"* e isso soará natural e coerente em nossa linguagem do mundo real. Agora *"Um(a) Economizar tem características e comportamentos..."* não soa tão bem, não concorda? Ou ainda, quando instanciamos um objeto da classe, o que soa mais natural? *"Eu instanciei um Cofrinho"* ou *"Eu instanciei um Economizar"*?

Nomes no **singular**, na maioria das vezes, fazem mais sentido para nomes de classes. Afinal, se temos uma classe `Pessoa` e a instanciamos, temos *uma pessoa*. Se temos uma classe `Investimento` e a instanciamos, temos *um investimento*. Se as classes fossem `Pessoas` e `Investimentos`, no plural, criaríamos *uma pessoas* ou *um investimentos*... Acha que ficaria bom assim?

Claro que nomes no singular não é uma regra absoluta. Vejamos uma classe `Configuracoes`. Supondo que cada atributo nela armazena uma configuração do sistema, podemos dizer que, ao instanciá-la, criamos *um objeto que tem as configurações do sistema*.

2.12 Nomes de métodos (ou funções)

Mais uma vez, convido você a olhar os códigos deste capítulo e observar os nomes dos métodos. Notou algo de comum entre eles? Notou que todos indicam uma *ação*?

Ora, o próprio conceito de métodos e funções é que são entidades que realizam uma ação e podem ou não retornar um valor para o que os invocou. Se um método ou função sempre realiza uma ação, nada mais claro que seu nome expresse isso.

Vamos imaginar que precisamos informar qual o maior valor depositado no cofrinho. Para tanto, criamos um método que retorna esse valor. Em uma primeira tentativa, faremos como nos exemplos a seguir.

Java:

```
public double maiorValorDepositado() {
    // corpo da função
}
```

Python:

```
def maior_valor_depositado():
    # corpo da função
```

Esses nomes não estariam de todo ruim, mas se o objetivo do método é nos entregar o maior valor que já foi depositado, melhor seria deixar isso explícito no nome. Assim, até mesmo um programador iniciante teria mais certeza do objetivo do método. Veja os próximos códigos.

Java:

```
public double obterMaiorValorDepositado() {
    // corpo da função
}
```

Python:

```
def obter_maior_valor_depositado():
    # corpo da função
```

2.13 Não seja engraçadinho

Senso de humor é uma característica fácil de notar em uma conversa informal, como em momentos de *happy hour* ou pausas para um cafezinho, por exemplo. Em momentos assim, normalmente é fácil notar quem é o gênio dos trocadilhos, quem é o rei das imitações e até quem é o "tio do pavê". Mas não deveria ser fácil notar isso ao ler o código-fonte de alguém. Portanto, se você tem um dom para o humor, porém prefere demonstrar profissionalismo em seus códigos, não caia na tentação de criar códigos como os seguintes:

Java:

```
private double jurosQueAmoSQN = 3.5;

public double byebyeMinhaGrana() {
```

```
    // corpo da função
}
```

Python:

```
juros_que_amo_sqn = 3.5

def byebye_minha_grana():
    # corpo da função
```

Esses nomes de variáveis e funções até seguem boa parte das recomendações já dadas neste capítulo, mas é muito pouco provável que sejam classificados como profissionais. Um código com tom profissional seria mais parecido com o que temos a seguir.

Java:

```
private double taxaJuros = 3.5;

public double sacarTudo() {
    // corpo da função
}
```

Python:

```
taxa_juros = 3.5

def sacar_tudo():
    # corpo da função
```

E, assim como não soa bem ser engraçadinho no código-fonte, igualmente se deve evitar ser ranzinza, irônico, racista ou preconceituoso. Foque no problema a ser resolvido com sua programação sendo o mais neutro e profissional possível.

2.14 Use termos técnicos de programação a seu favor

Imagine um projeto médio/grande onde todas as classes são somente substantivos e adjetivos, tais como `Cofrinho`, `Investimento`, `Conexao`, `Usuario`, `Amigo` etc. Bom, infelizmente isso vai ficar só no campo da imaginação mesmo. Isso porque, em um projeto médio ou maior, os nomes de classes terão que, eventualmente, conter termos que expliquem o que uma entidade faz.

Vamos supor que nosso cofrinho evolua para um projeto que acesse banco de dados e ofereça uma REST API (se não sabe o que é uma, fique na paz, apenas observe os exemplos de nomes a seguir). Em vez de reinventar a roda, vamos usar **técnicas consolidadas**, o que facilita pesquisas em livros/internet para programar mais rápido e também torna mais fácil a entrada de novos membros na equipe do projeto.

Mãos à obra: existe a necessidade de uma tabela de banco de dados chamada "cofrinho". Para criar a API, decidimos usar a técnica *ORM* (*Object Relational Mapping*, ou *Mapeamento Objeto-Relacional* em tradução livre), na qual criamos uma classe para mapear uma tabela. Decidimos também usar um padrão de projeto chamado *Repository*, no qual é criada uma classe que contém métodos de acesso ao banco de dados para uma determinada tabela.

Nesse caso, a classe que mapeia a tabela "cofrinho" poder ser simplesmente `Cofrinho` e a classe *Repository* pode se chamar `CofrinhoRepository`. Por fim, a classe que vai conter as chamadas da API pode se chamar `CofrinhoController` ou `CofrinhoResource`, ambos os padrões de nomes mais usados para classes com esse tipo e finalidade.

Usando esses nomes, nossas pesquisas em livros/internet serão facilitadas, além de que qualquer desenvolvedor que tem o mínimo de experiência com REST API saberia a finalidade de cada uma dessas classes, visto que foram usados nomes que seguem **convenções**.

Até mesmo em variáveis e funções podemos aplicar **técnicas consolidadas** e **convenções**. Por exemplo, criamos várias funções que começam com o termo `obter` neste capítulo, certo? Ficaram bons nomes, porém, existe uma convenção, um padrão chamado *Get/Set*. Segundo ele, quando criamos funções/métodos que retornam um valor, seu nome deve começar com `get` ou, se for um valor booleano (*true*/*false*), pode ser também `is`. Se a função/método tem como objetivo atribuir algum valor, seu nome deve começar com `set`. Seguindo esse padrão, poderíamos reescrever nossos `obter???()` como nos códigos a seguir.

Java:

```
String getDono() {
    // corpo da função
}

String getNomeCofrinho() {
    // corpo da função
}

String getObjetivo() {
    // corpo da função
}
```

Python:

```
def get_dono(self):
    # corpo da função

def get_nome_cofrinho(self):
    # corpo da função

def get_objetivo(self):
    # corpo da função
```

Um caso especial de *Get* é o de um método retornar um valor booleano quando usamos o prefixo `is` (que quer dizer se algo *existe* ou *é*). Veja os próximos exemplos.

Java:

```java
boolean isCheio() {
    // retorna true se o cofrinho está cheio ou false em caso contrário
}
```

Python:

```python
def is_cheio(self):
    # retorna true se o cofrinho está cheio ou false em caso contrário
```

Ainda seguindo esse padrão, poderíamos reescrever nossos `definir???()` usando `set`, como nos códigos a seguir.

Java:

```java
void setNomeDonoCofrinho(String nome) {
    // corpo da função
}

void setNomeCofrinho(String nome) {
    // corpo da função
}

void setNomeObjetivoCofrinho(String nome) {
    // corpo da função
}
```

Python:

```python
def set_nome_dono_cofrinho(nome):
    # corpo da função

def set_nome_cofrinho(nome):
    # corpo da função

def set_nome_objetivo_cofrinho(nome):
    # corpo da função
```

2.15 Não adicione termos desnecessários e repetitivos

Imagine que seu programa de cofrinho ficou bem legal e decidiu colocá-lo em uma loja de aplicativos para vender ou ganhar dinheiro com propagandas nele. Claro que precisa dar nome à sua criação, certo? Daí você resolveu chamá-lo de **Cofreedom**. Até aí, tudo bem... Mas eis que surgiu a grande ideia de pôr o nome do aplicativo no nome de todas as classes. Assim, nossas classes passariam a se chamar: `CofreedomCofrinho`, `CofreedomCofrinhoRepository`, `CofreedomCofrinhoController` e assim por diante.

Fazer esse tipo de coisa, não acrescenta nada de positivo no cotidiano da evolução e manutenção do código-fonte. Ao contrário, traz complicações, dentre as quais destaco:

- Os nomes de todas as classes serão inevitavelmente e desnecessariamente grandes
- A pesquisa por entidades na IDE ficará mais difícil, uma vez que toda classe contém a mesma palavra
- Serão maiores as chances de se mexer na classe errada, afinal, várias letras serão iguais no nome de todas
- Programadores serão menos felizes. Afinal, quem ficaria feliz em digitar `Cofreedom` sempre que for digitar o nome de uma nova classe?

Este capítulo foi extenso e intenso, mas é o coração das orientações de Código Limpo. Apesar de serem muitas as orientações sobre nomes, todas partem da reflexão sobre os seguintes questionamentos:

- O nome desta entidade (classe, variável, função, parâmetro etc.) revela facilmente a intenção dela?
- O nome poderia seguir alguma convenção?
- Esse nome me trará algum problema que não seja de cunho técnico?
- Esse nome vai facilitar a vida dos atuais e novos colegas que terão contato com o código?

No próximo capítulo, veremos como seriam funções bem escritas em um código limpo.

Capítulo 3
Funções

Se alguém pergunta as horas para o João Kacau, ele saberá informar isso facilmente caso esteja com um relógio ou um celular, certo? Já parou para pensar quantas ações ele precisa realizar para poder dizer para outra pessoa que horas são? Se considerarmos que vai ver as horas em um relógio de pulso, seriam mais ou menos estas:

1. Lembrar o braço em que está o relógio;
2. Movimentar o braço para uma posição na qual ele possa ver a tela do relógio;
3. Olhar para os números do relógio;
4. Traduzir a informação na tela (números ou posição dos ponteiros) para uma "hora";
5. Informar o valor da hora atual usando o valor que traduziu na ação anterior.

Cada uma dessas cinco ações só é feita por ele quando for informar a hora para alguém? Creio que não, afinal ele pode querer lembrar onde está o relógio apenas para escondê-lo ou para escolher um braço que ficará sujo em uma determinada ação. Pode movimentar o braço para uma posição na qual possa ver a tela do relógio para ver que dia é hoje, fazendo a tradução dos números de forma diferente da que usou para ler as horas. E ele pode informar o valor da hora atual usando um valor de hora que ouviu há poucos segundos de outra pessoa ou na rádio, por exemplo.

Em um programa de computador também é assim. Um clique em um botão da tela solicita a execução de uma função que pode disparar outras funções, e estas podem disparar outras, e assim por diante. **Uma função pode ser entendida com um miniprograma que realiza uma ação específica**.

Existe um espaço dedicado apenas para falar de funções nas técnicas de Código Limpo por um motivo: muitos programadores criam funções que até funcionam, mas de uma maneira muito ruim e, às vezes, até perigosa. Porém, antes de continuarmos no que diz respeito às práticas de Código Limpo em relação às funções, gostaria de deixar clara a diferença entre os conceitos **função**, **método** e **procedimento** nos quadros a seguir.

Função (ou *function*): é um bloco de código que pode ser invocado pelo seu nome. No seu conceito original, que ainda é encontrado em muitos livros, se executada sem erros, uma função retorna um valor. Porém, em várias linguagens de programação modernas como JavaScript, Python e Groovy, uma função não precisa retornar um valor. Uma função pode ou não precisar de informações para funcionar, as quais chamamos de **parâmetros**.

Procedimento (ou *procedure*): é quase o mesmo que função. A única diferença é que um procedimento não retorna valor algum quando executado. A não ser que você programe em bancos de dados relacionais, raramente vai usar o termo *procedimento* (ou *procedure*). Nos bancos de dados, eles são comumente chamados de ***stored procedures***. Nas linguagens de programação de uso geral (Java, JavaScript, Python, Kotlin, Groovy etc.), raramente uma função sem retorno é chamada de procedimento; só se chama de *função* ou *método*.

MÉTODO (OU *METHOD*): é basicamente o mesmo que uma função e um procedimento, porém está fisicamente dentro de uma classe em um código de uma linguagem que implementa o paradigma da *POO (Programação Orientada a Objetos)*. Ou seja, todo método é uma função. Uma função dentro de uma classe *continua sendo uma função*, porém também pode ser chamada de *método*.

3.1 Funções devem resolver um problema bem específico

Quando estamos dando os primeiros passos em programação, é comum criar um programa no qual um único bloco de código implementa tudo o que o programa faz. Só que, quando se aprende a criar funções, passa a ser muito importante dividir o programa em várias funções, para que o código fique mais organizado, mais fácil de testar e de dar manutenções corretivas e evolutivas.

Certa vez, conversando com um aluno sobre esse tema, ele me perguntou se não ficava ruim o código de um programa ter centenas de pequenas funções. Eu disse a ele e reafirmo aqui que **é muito melhor ter um programa cheio de pequenas funções do que com poucas funções gigantes**.

Quando uma função é bem específica, dizemos que ela tem **alta coesão**. Quanto mais coisas diferentes uma função faz, mais baixa é sua coesão. Um código limpo exige funções com alta coesão.

OPINIÃO PROFISSIONAL

"Considero imperdoável ter funções muito grandes. É necessário que elas sejam cada vez menores, para que possamos reutilizá-las mais vezes e, por consequência, escrever menos código a médio/longo prazo."

Rodrigo Vieira Pinto, mestre em Engenharia de Software pelo IPT-SP.

Vejamos o seguinte código de uma função que calcula o desconto de VT (vale-transporte) sobre um salário. As regras são:

- Uma passagem custa R$ 4,50;
- O valor total do desconto de VT não pode ultrapassar 6% do valor do salário;
- O valor do salário deve ser maior ou igual a um salário mínimo, que é de R$ 1.000,00;
- Considere que o trabalhador trabalha 22 dias por mês;
- Para funcionar, a função recebe a quantidade de passagens que o trabalhador precisa para ir e voltar do trabalho e o valor de seu salário.

Java:

```
Double getValorVT(Integer passagensCasaTrabalho, Double salario) {

    if (salario < 1000) {
        return null;
    }

    var totalPassagensMes = passagensCasaTrabalho * 2 * 22;
    var vtMes = totalPassagensMes * 4.5;
    var valorLimite = salario * 0.06;

    if (vtMes > valorLimite) {
        return valorLimite;
```

```
    } else {
        return vtMes;
    }

}
```

Python:

```
def get_valor_VT(passagens_casa_trabalho, salario):

    if (salario < 1000):
            return None

    total_passagens_mes = passagens_casa_trabalho * 2 * 22
    vt_mes = total_passagens_mes * 4.5
    valor_limite = salario * 0.06

    if (vt_mes > valor_limite):
        return valor_limite
    else:
        return vt_mes
```

Os códigos implementam as regras do nosso cálculo. Porém, note que a função realiza coisas demais. Pense, por exemplo, que surja a necessidade de uma outra função que calcula algum outro desconto relacionado a salário. Muito provavelmente precisaremos validar o salário mínimo nela. E se não forem mais 22 dias por mês, e sim 21? Ou, ainda, e se precisarmos do valor da passagem em outra função? Por fim, imagine que o limite de valor de VT mude, o que você faria? Vejamos como o código poderia ficar mais limpo dividindo a função em várias, como nos códigos a seguir.

Java:

```
boolean isSalarioValido(Double salario) {
   return salario >= getSalarioMinimo();
}

Double getSalarioMinimo() {
   return 1000.0;
```

```
}

Integer getDiasUteisMes() {
   return 22;
}

Double getValorPassagem() {
   return 4.5;
}

Double getPorcentagemLimiteVt() {
   return 6.0;
}

Double getValorVT(Integer passagensCasaTrabalho, Double salario) {

   if (isSalarioValido(salario)) {
      var totalPassagensMes = passagensCasaTrabalho * 2 *
getDiasUteisMes();
      var vtMes = totalPassagensMes * getValorPassagem();

      var valorLimite = salario * getPorcentagemLimiteVt() / 100;

      if (vtMes > valorLimite) {
         return valorLimite ;
      } else {
         return vtMes;
      }
   } else {
      return null;
   }

}
```

Python:

```
def is_salario_valido(salario):
  return salario >= get_salario_minimo()

def get_salario_minimo():
```

```
  return 1000.0

def get_dias_uteis_mes():
  return 22

def get_valor_passagem():
  return 4.5

def get_porcentagem_limite_vt():
  return 6.0

def get_valor_VT(passagens_casa_trabalho, salario):

  if (is_salario_valido(salario)):

      total_passagens_mes = passagens_casa_trabalho * 2 *
get_dias_uteis_mes()
      vt_mes = total_passagens_mes * get_valor_passagem()
      valor_limite = salario * get_porcentagem_limite_vt() / 100

      if (vt_mes > valor_limite):
         return valor_limite
      else:
         return vt_mes

      else:
         return None
```

Ficou claro que elevamos muito a quantidade de funções. Fomos de uma para seis. Mas o que ganhamos com isso? Outras funções de cálculo que envolvem salário podem reaproveitar a função de validação de salário bem como os valores de salário mínimo, dias úteis no mês, passagem e porcentagem limite de VT. Se olharmos apenas para a função que retorna o valor de desconto de VT, percebemos que ela não se preocupa mais com valores que ela precisa, uma vez que os recupera ou dos parâmetros ou das outras funções. É como se tivéssemos criado um Lego™ de funções.

3.2 Nossa, quantos parâmetros você tem!

Para que funções tenham o código limpo, elas devem ter o mínimo de parâmetros possível. Para você ter uma ideia, o número considerado ideal para a quantidade de parâmetros em uma função é, literalmente, zero. Deve haver um motivo muito forte para que uma função tenha dois ou mais parâmetros, pois, em um código limpo, a maioria das funções com parâmetros vai esperar por apenas um parâmetro.

PARÂMETROS X ARGUMENTOS

Existem dois termos que costumam gerar confusão quando falamos de funções: **parâmetros** e **argumentos**.

Parâmetros: são aqueles que estão no código-fonte da função, em sua assinatura. Veja a função de exemplo a seguir.

Java:

```
Boolean isSalarioValido(Double salario) {
   // código da função
}
```

Python:

```
def is_salario_valido(salario):
   # código da função
```

Ela possui um **parâmetro**, que é `salario`.

Argumentos: são os valores que passamos em uma função ao invocá-la. Veja os códigos de invocação da função a seguir.

Java:

```
var resultado = isSalarioValido(1000.0);
```

Python:

```
resultado = is_salario_valido(1000.0)
```

O valor `1000.0` é o **argumento** usado para invocar a função.

Caso uma função realmente precise de dois ou mais parâmetros, a recomendação é criar uma classe e usar uma instância dela como um ***Argument Object*** (*objeto de argumento*, em tradução livre).

Vejamos o exemplo da função a seguir, que teria como objetivo calcular o benefício auxílio-creche.

Java:

```java
import java.time.LocalDate;

public Double getAuxilioCreche(String funcionario, Double salarioBase,
Integer filhos, LocalDate dataAdmissao) {
    // código que usa os valores de quatro parâmetros
}
```

Python:

```python
from datetime import date

def get_auxilio_creche(funcionario: str, salario_base: float, filhos: int,
data_admissao: date) -> float :
    # código que usa os valores de quatro parâmetros
```

Para que o código da função de cálculo de auxílio-creche fique mais limpo, vamos fazê-la esperar por apenas um parâmetro, em vez de quatro. Para isso, criaremos uma classe `ParametrosAuxilioCreche` para que possamos usar um *Argument Object*. Segue o código dela.

Java:

```java
import java.time.LocalDate;

public class ParametrosAuxilioCreche {
    private String funcionario;
    private Double salarioBase;
    private Integer filhos;
    private LocalDate dataAdmissao;

    // construtor(es) e getters/setters
}
```

Python:

```python
from datetime import date

class ParametrosAuxilioCreche:

    def __init__(self, funcionario: str, salario_base: float, filhos: int,
data_admissao: date) :
        self.funcionario = funcionario
        self.salario_base = salario_base
        self.filhos = filhos
        self.data_admissao = data_admissao

    # getters/setters
```

Assim, a função de cálculo de auxílio-creche poderia ficar como nos próximos códigos.

Java:

```java
public Double getAuxilioCreche(ParametrosAuxilioCreche parametros) {
    // código que usa os valores do parâmetro único
}
```

Python:

```python
def get_auxilio_creche(parametros: ParametrosAuxilioCreche) -> float :
    # código que usa os valores do parâmetro único
```

Uma importante vantagem dessa nova abordagem é que podemos abstrair regras de validação dos parâmetros na classe `ParametrosAuxilioCreche`, então a função de cálculo de auxílio-creche teria como responsabilidade apenas realizar os cálculos, sem se preocupar com validações de parâmetros.

3.3 Evite flags

Funções com código limpo não deveriam possuir parâmetros do tipo ***flag***. São parâmetros booleanos que fazem com que a função tenha

dois comportamentos bem diferentes para `verdadeiro` e para `falso`. Um exemplo de função que não está com código limpo devido ao uso de parâmetro *flag* seria uma função de cálculo de adicional de férias de um funcionário em que teríamos duas possibilidades de cálculo bem diferentes entre si. Vejamos no código a seguir.

Java:

```
public Double getAdicionalFerias(Double salarioBase, Boolean vendeuFerias)
{
    if (vendeuFerias) {
        // código que faz os cálculos para o caso de venda de férias, bem
diferente do cálculo no bloco 'else' a seguir
    } else {
        // código que faz os cálculos para o caso em que não há venda de
férias, bem diferente do cálculo no bloco 'if' anterior
    }
}
```

Python:

```
def get_adicional_ferias(salario_base: float, vendeu_ferias: bool) ->
float :
    if (vendeu_ferias):
        # código que faz os cálculos para o caso de venda de férias, bem
diferente do cálculo no bloco 'else' a seguir
    else:
        # código que faz os cálculos para o caso em que não há venda de
férias, bem diferente do cálculo no bloco 'if' anterior
```

Devido ao parâmetro *flag*, a função de adicional de férias possui, na verdade, duas versões: uma para o *flag* sendo `verdadeiro`, outra para `falso`. Como esse parâmetro fez com que a função passasse a ter pouca coesão, o ideal seria eliminá-lo e dividir a função em duas, como no exemplo a seguir.

Java:

```
public Double getAdicionalFeriasComFeriasVendidas(Double salarioBase) {
    // código que faz os cálculos para o caso de venda de férias
```

```
}

public Double getAdicionalFerias(Double salarioBase) {
    // código que faz os cálculos para o caso em que não há venda de
férias
}
```

Python:

```
def get_adicional_ferias_com_ferias_vendidas(salario_base: float) -> float
:
    # código que faz os cálculos para o caso de venda de férias

def get_adicional_ferias(salario_base: float) -> float :
    # código que faz os cálculos para o caso em que não há venda de férias
```

Agora temos duas funções, ambas com alta coesão, pois cada uma ficou focada apenas em seu cenário.

3.4 Listas como argumentos

Existem situações em que uma função pode receber vários parâmetros (bem mais que quatro, por exemplo) e mesmo assim isso estar tudo bem. Normalmente, são situações em que os valores dos argumentos recebidos não influenciarão no comportamento, na lógica da função.

As linguagens de programação mais usadas da atualidade possuem um recurso que permite que uma função receba de 0 a N valores para um parâmetro, tratando-o internamente como um vetor ou lista. Em Java, podemos usar o recurso de **varargs** ou simplesmente usar um vetor ou alguma **Collection** (`List` , `Set` etc.) como parâmetro. Em Python, podemos usar o recurso de ***args** ou ****kwargs**.

Um exemplo seria a função que cria um texto formatado com todos os descontos de um salário, como nos códigos a seguir.

Java, usando **varargs**:

```java
public String getTextoDescontos(Double... descontos) {
    var texto = new StringBuilder("Descontos: \r\n");
    var total = 0.0;

    for (var i = 0; i < descontos.length; i++) {
        texto.append((i+1)+" -> " + descontos[i] + "\r\n");
        total += descontos[i];
    }
    texto.append("Total de descontos: R$" + total);

    return texto.toString();
}

// para invocar a função, seria como nestes exemplos:
getTextoDescontos(150.0, 25.0, 89.90);
getTextoDescontos(350.50);
getTextoDescontos(77.0, 15.41);
getTextoDescontos();
```

Python, usando ***args**:

```python
def get_texto_descontos(*descontos):
    texto = 'Descontos: \r\n'
    total = 0

    for i in range(len(descontos)):
        texto += f'{str(i+1)} -> {str(descontos[i])} \r\n'
        total += descontos[i]

    texto += f'Total de descontos: R${str(total)}';

    return texto

# para invocar a função, seria como nestes exemplos:
get_texto_descontos(150.0, 25.0, 89.90)
```

```
get_texto_descontos(350.50)
get_texto_descontos(77.0, 15.41)
get_texto_descontos()
```

Na função recém-descrita, é possível passar de 0 a N valores como `descontos`, porém nem a quantidade nem a ordem vão influenciar no comportamento da função. Ela simplesmente faz um cálculo simples de soma e monta um texto usando todos os valores recebidos.

A ordem dos descontos também não importa, pois não há nenhuma menção se eles deveriam estar em ordem cronológica ou de valor monetário ou em qualquer outra ordem. É bem diferente de uma função, por exemplo, de transferência bancária, cujos parâmetros seriam *de quem* e *para quem*. Nesse caso, a inversão dos valores poderia comprometer seriamente o resultado da execução da função.

3.5 Surpresa! Eu odeio surpresas!

Se no mundo real surpresas são alvos controversos, pois nem sempre são bem recebidas, por melhor que seja a intenção, quando se trata de funções elas deveriam ser inadmissíveis! Uma função **nunca** deveria provocar um efeito colateral, ou seja, não esperado, de forte consequência. Vejamos no código de validação de salário a seguir.

Java:

```java
public Boolean isSalarioValido(Double salario) {
    if (salario > 8000) {
        demitirFuncionario();
        return false;
    }

    Boolean valido = // algum código para validar
```

```
    return valido;
}
```

Python:

```
def is_salario_valido(self, salario) :
    if (salario > 8000) :
        demitir_funcionario()
        return False

    valido = # algum código para validar
    return valido
```

Note que uma função cujo nome indica que apenas vamos validar um salário pode simplesmente **demitir** um funcionário! Isso seria um efeito colateral oculto bem mal implementado. Se existir essa regra de demitir quando o salário for maior que 8 mil, que ela seja implementada em outro lugar. Do jeito que está, nós não podemos, por exemplo, fazer uma segunda validação de salário em um funcionário, uma vez que ele já foi demitido na primeira.

3.6 Faça algo OU conte-me algo

Muito já foi dito aqui sobre bons nomes e sobre o fato de uma função bem escrita fazer uma única coisa. Por isso, devemos evitar cair no erro de criar uma função que, ao mesmo tempo em que realiza uma ação, nos conta algo, mesmo que seja sobre o resultado da ação.

Veja, não estou dizendo que funções não devem ter retorno. Se uma função tem como objetivo obter o valor de desconto de VT de um salário, é claro que ela deve ter retorno. Se uma função tem como objetivo validar um salário, é claro que ela deve ter retorno. Mas, e se uma função tem como objetivo transferir um dinheiro para

alguém? Ela deve ter retorno? Se sim, o que seria? Vejamos um exemplo da função `transferir`.

Java:

```
public Boolean transferir(Double valor, Integer idDestinatario) {
    // retorna true se transferiu ou false, caso contrário
}
```

Python:

```
def transferir(valor, id_destinatario) {
    # retorna true se transferiu ou false, caso contrário
}
```

Em um primeiro olhar, essa função parece estar OK, certo? O nome condiz com seu objetivo. Há apenas dois parâmetros. Mas ela retorna um booleano que indica se houve ou não a transferência. Isso faz a função ter dois problemas:

1. Quem a usa pode acabar optando por usá-la em um teste lógico (algo como `if (transferir(1500.50, 2)`)
2. Em caso de a transferência não ter sido efetivada, não sabemos o motivo. Só sabemos que "porque não".

É por situações como a da função `transferir()` que devemos evitar que uma função **faça** algo e, ao mesmo tempo, **nos conte** algo. O ideal seria a função lançar uma exceção para o caso de não conseguir realizar a transferência. O código ficaria bem limpo e fácil de descobrir e tratar o motivo da falha na operação.

Portanto, a função `transferir()` deveria ser refatorada para se tornar uma função sem retorno, porém com lançamento de exceção para o caso de falha na transferência. Mais detalhes e exemplos desse tipo de técnica são mostrados no capítulo 7, sobre tratamento de exceções.

3.7 DRY — Don't Repeat Yourself (Não repita a si mesmo)

Há pequenos problemas que só surgem com o crescimento do código. Um deles é a repetição de código entre funções. É comum acontecer algo como:

1. Alguém implementa um código dentro de uma função;
2. A mesma rotina implementada na função anterior precisa ser feita em outra função. Caso a pessoa responsável por codar essa outra função não tenha ciência de que está repetindo um código, passaremos a ter um potencial problema devido a essa repetição. Mas, se ela perceber a duplicidade, deveria ocorrer um terceiro passo, que seria...;
3. A rotina necessária nas funções dos passos 1 e 2 é isolada em uma outra função, sendo invocada sempre que necessário.

Para ilustrar como seria essa situação, vejamos o exemplo de duas funções: uma que calcula o auxílio-creche e outra que calcula o bônus de final de ano. Ambas recebem como argumento a quantidade de filhos que o funcionário tem; além disso, ambas possuem uma regra em comum: a quantidade de filhos deve ser entre 0 (zero) e 4 (quatro). Vejamos o código a seguir.

Java:

```
public Double getAuxilioCreche(Integer filhos) {
    if (filhos >= 0 && filhos <= 4) {
        // cálculo do auxílio-creche
    } else {
        // código para o caso de a quantidade de filhos ser inválida
    }
}

public Double getBonusAnual(Integer filhos) {
    if (filhos >= 0 && filhos <= 4) {
```

```
        // cálculo do bônus
    } else {
        // código para o caso de a quantidade de filhos ser inválida
    }
}
```

Python:

```
def  get_auxilio_creche(filhos) :
    if (filhos >= 0 and filhos <= 4) :
        # cálculo do auxílio-creche
    else :
        # código para o caso de a quantidade de filhos ser inválida

def  get_bonus_anual(filhos) :
    if (filhos >= 0 and filhos <= 4) :
        # cálculo do bônus
    else :
        # código para o caso de a quantidade de filhos ser inválida
```

Percebeu como o `if` ficou repetitivo nos códigos? Isso vai além da questão estética. Imagine se não fossem só duas, mas três, quatro, cinco repetições; ou se nós precisássemos mudar a regra para passar a aceitar só até três filhos. Pense em quanto lugares precisaríamos alterar nosso código. Isso, além de ser trabalhoso, é perigoso. E se esquecermos em apenas um lugarzinho? É para evitar isso que devemos usar o princípio **DRY** (***Don't Repeat Yourself***, "não repita a si mesmo", em tradução livre). Nesse caso, isolaríamos a validação da quantidade de filhos em uma função à parte. Nosso código ficaria como o a seguir.

Java:

```
public Double getAuxilioCreche(Integer filhos) {
    if (isFilhosValidos(filhos)) {
        // cálculo do auxílio-creche
    } else {
        // código para o caso de a quantidade de filhos ser inválida
    }
```

```
}

public Double getBonusAnual(Integer filhos) {
    if (isFilhosValidos(filhos)) {
        // cálculo do bônus
    } else {
        // código para o caso de a quantidade de filhos ser inválida
    }
}

public Boolean isFilhosValidos(Integer filhos) {
    return filhos >= 0 && filhos <= 4;
}
```

Python:

```
def  get_auxilio_creche(filhos) :
    if (is_filhos_validos(filhos)) :
        # cálculo do auxílio-creche
    else :
        # código para o caso de a quantidade de filhos ser inválida

def  get_bonus_anual(filhos) :
    if (is_filhos_validos(filhos)) :
        # cálculo do bônus
    else :
        # código para o caso de a quantidade de filhos ser inválida

def is_filhos_validos(filhos) :
    return filhos >= 0 and filhos <= 4
```

Note que o código ficou mais limpo, pois repetimos apenas a invocação da função, mas não sua lógica. Se mudarmos a regra do que vem a ser uma quantidade válida de filhos, basta mudar em um lugar, que é na função que valida a quantidade de filhos.

Neste capítulo, vimos que devemos pensar bem nos parâmetros e na coesão das funções. No próximo, veremos que um código limpo

também depende do que não é necessariamente código, que são os comentários de código.

Capítulo 4
Comentários de código

Certo dia, João Kacau estava visitando a casa de uma amiga. Chegando lá, notou que havia um tipo de bilhete colado em todos os objetos dizendo para o que aquilo servia. No sofá, por exemplo, o bilhete dizia "Isto é um sofá. Nele, você pode sentar e deitar". No bilhete afixado na geladeira, constava "Isto é uma geladeira. Nela, você guarda coisas que quer que fiquem geladas". João achou aquilo estranho, mas ainda brincou com a amiga dizendo que, no bilhete do sofá, poderia acrescentar que não se pode sentar ou deitar com roupas molhadas. O problema é que ela levou a sério e acrescentou essa observação ao texto do sofá.

A amiga de João disse que fez aquilo para que ninguém que a visitasse ficasse com dúvida sobre o objetivo dos objetos ou como usá-los. João então explicou que a imensa maioria dos objetos não precisava daquilo, porque o uso dos objetos ou era de conhecimento comum (como o sofá ou a geladeira), ou era de uso bem intuitivo, como o controle da TV, por exemplo. João concordou que apenas pouquíssimos objetos tivessem alguma instrução, como a churrasqueira elétrica, por exemplo, porque era cheia de funcionalidades e de uma marca importada, pouco conhecida no país.

Se você achou estranho o que a amiga de João fez, já parou para pensar que pode estar fazendo algo parecido em sua programação? Se um código-fonte contém muito comentário sobre o que ele faz e como deve ser usado, é como uma casa com bilhetinhos nos objetos.

4.1 O que são comentários de código

Comentários de código, ou simplesmente *comentários*, são trechos no código que não influenciam na execução do programa. Podem ser usados para explicar o que determinado trecho de código faz, para orientar como invocar uma função, para explicar o que uma função pode retornar, para anotações pessoais de quem está programando etc.

A maioria das linguagens de programação possui dois tipos de comentários: **simples** (ou **de uma linha**) e **de Bloco**. O *comentário simples* ocupa parte de uma linha ou uma linha inteira. O *comentário de bloco* ocupa mais de uma linha.

Vamos ver alguns exemplos de comentários simples. Em linguagens como Java, JavaScript, Groovy e Kotlin, um comentário simples começa com `//` :

```
// esta linha não influencia no código
String bairro = "Jurunas";  // esta linha possui um código "de verdade" e um comentário também
```

Em *Python*, um *comentário simples* começa com `#` :

```
# esta linha não influencia no código
bairro = "Jurunas";  # esta linha possui um código "de verdade" e um comentário também
```

Agora, vamos ver exemplos de *comentário de bloco*. Em linguagens como Java, JavaScript, Groovy e Kotlin, um comentário de bloco começa com `/*` e termina com `*/` :

```
/*
 este bloco de comentário
 não influencia no código
*/
String time = "Remo";
```

Em *Python*, um comentário de bloco começa e termina com `"""` (três aspas duplas):

```
"""
 este bloco de comentário
 não influencia no código
"""
time = "Remo";
```

Algumas linguagens, como o HTML e o CSS, usam os mesmos delimitadores para comentários simples e de bloco. Vamos ver alguns exemplos. Em HTML, um comentário começa com `<!--` e termina com `-->`:

```
<!--
 este bloco de comentário
 não influencia no código
-->
<div>Conteúdo da notícia</div>
<!-- esta linha não influencia no código -->
```

Em CSS, um comentário começa com `/*` e termina com `*/`:

```
div {
  color: red;
  /* esta linha não influencia no código */
}
/*
 este bloco de comentário
 não influencia no código
*/
```

4.2 Comentários ajudam a explicar o código?

Vamos resgatar as funções de verificação de salário válido e de salário mínimo do capítulo anterior:

Java:

```
Boolean isSalarioValido(Double salario) {
    return salario >= getSalarioMinimo();
}

Double getSalarioMinimo() {
    return 1000.0;
}
```

Python:

```
def is_salario_valido(salario):
    return salario >= get_salario_minimo()

def get_salario_minimo():
    return 1000.0
```

Convido você a refletir se esses métodos precisam de comentários explicando o que eles fazem e como invocá-los. Olhe para o código todo, desde sua assinatura até seu corpo. Mas, se realmente achar que comentários caem bem, talvez eles fiquem assim:

Java:

```
/*
Retorna true se o parâmetro 'salario' for um salário válido, que no caso é
maior ou igual a 1 salário mínimo
*/
Boolean isSalarioValido(Double salario) {
    return salario >= getSalarioMinimo();
}

// retorna o valor do salário mínimo, que é 1000.0
Double getSalarioMinimo() {
    return 1000.0;
}
```

Python:

```
"""
Retorna true se o parâmetro 'salario' for um salário válido, que no caso é
```

```
maior ou igual a 1 salário mínimo
"""
def is_salario_valido(salario):
    return salario >= get_salario_minimo()

# retorna o valor do salário mínimo, que é 1000.0
def get_salario_minimo():
    return 1000.0
```

Você acha que o código ficou mais claro agora? E se a regra de um salário válido for que ele pode ser a partir de um salário mínimo + 5%? E mais: o salário mínimo agora é de `2000.0`. Onde você deveria mexer no código para que essas novas regras passassem a valer? No código-fonte que é realmente compilado ou nos comentários?

Talvez você pense que basta mudar sempre em ambos: no código e no comentário. Sim, seria interessante, mas perceba que se você atualizar somente o código, que é o que realmente influencia no programa, a nova regra estará de fato valendo. Assim teremos algo que dá uma informação errada, que são os comentários desatualizados. Isso porque as **mudanças no código-fonte não provocam mudanças nos comentários**. Assim, é possível concluir que não existe **nenhuma** garantia de que o comentário realmente explique o que o código-fonte faz e/ou como usá-lo em outros códigos.

É por isso que a recomendação de Código Limpo é que o código seja o mais claro possível, para que não exista a necessidade de comentários para explicá-lo. Sabe aquela sensação de "*nossa, quem fez isso é um gênio*" que você sente quando usa um objeto, brinquedo, assessório ou aplicativo que é superfácil de usar sem ler manual nem perguntar a ninguém? É essa a sensação que o nosso código-fonte deve causar em quem ler e/ou usar o que programamos.

O código não está muito bom, alguns comentariozinhos não deixariam o código mais fácil de entender? **Não!** Melhore os nomes, reduza o tamanho das funções e/ou das classes. Os comentários *não devem ser* o que deixa o código mais fácil de entender!

4.3 Um código limpo não pode ter nenhum comentário?

Um código bem escrito pode conter comentários, mas apenas para situações em que o código-fonte realmente pode não deixar claro o que está acontecendo. Exemplos disso são:

- **Informações autorais**: identificação do(s) autor(es) do código, incluindo informações de licença e propriedade intelectual;
- **Comentários de métodos abstratos**: métodos abstratos de classes abstratas e interfaces às vezes podem precisar de um comentário para que fique mais claro o seu objetivo;
- **Exemplos de expressões regulares**: caso uma expressão regular (ou *regex*) seja fundamental no entendimento de um trecho de código, um comentário com um ou mais exemplos do que se espera na expressão pode ser útil;
- **Explicar decisões de código**: caso uma determinada decisão de código tenha sido tomada para melhoria de desempenho ou devido a um *bug* em uma biblioteca, é interessante acrescentar um comentário;
- **Quem avisa, amigo é**: se uma função puder causar lentidão ou até mesmo travar um sistema, é bom isso estar avisado em forma de comentário.

4.4 Sou dev iniciante, por isso comento todo o meu código. E agora?

Como leciono há anos, já vi várias vezes códigos de pessoas novatas em programação repletos de comentários (às vezes quase um comentário por linha). Nesse caso, eu não critico os alunos; ao contrário, os parabenizo. Nessa situação, entendo que o código-fonte ainda não é de um programa "de verdade", e sim um exercício, uma prática de alguém que está dando os primeiros passos na programação. Portanto, se esse tipo de comentário ajudar o estudante a aprender mais a nobre arte da programação, por que não?

4.5 Comentários com palavras-chave (TODO, FIXME)

Algumas palavras-chave acabaram se tornando convenções para comentários, ajudando a classificar o que está descrito. As principais palavras-chave de classificação de comentários são:

- **TODO**: "A fazer", em tradução livre. Usamos quando queremos escrever no comentário que falta algo a ser feito no código.
- **FIXME**: "Corrija-me", em tradução livre. Usamos quando queremos escrever no comentário que uma correção precisa ser feita no código.

ATENÇÃO: Esses termos devem ser usados com moderação! Só inclua esse tipo de comentário se for sua *última opção* — se já estiver na hora de encerrar o expediente ou se não souber como implementar o que deve ser feito, por exemplo.

4.6 Comentários para documentar o código

É possível usar técnicas e ferramentas que usam comentários para documentar um código. Usamos esse tipo de ferramenta principalmente quando transformamos nosso código em uma biblioteca para que seja reaproveitada por outros desenvolvedores.

Nesse caso, devemos usar as ferramentas de documentação de cada linguagem de programação. No caso de Java, é a **Javadoc** e, no caso de Python, é a **Docstring** (ou **PEP 257**). Usando essas ferramentas, é possível gerar documentações padronizadas em formatos como HTML e PDF.

4.7 Sem comentários!

Existem coisas que realmente não fazem muito sentido estarem como comentários no código, dentre elas:

- Métodos de comportamento padrão, como *getters*, *setters*, *equals* e construtores padrão;
- Detalhes de versão de código (quem fez o que e quando). Para isso, existem as mensagens de *commit* de sistemas de gerenciamento de versão como o Git e o Mercurial;
- Elogios, críticas, piadinhas, dúvidas e desabafos. Para isso, use reuniões, e-mails, calls etc.

OPINIÃO PROFISSIONAL

"Comentários são necessários e bem-vindos, porém é comum encontrar projetos que fazem uso deles com pouco ou nenhum propósito, como aqueles comentários em métodos *set* e *get* que não contribuem em nada para o contexto geral. Há quem defenda que um código bem escrito não precisa de comentários; porém, acredito que existem, sim, situações em que um comentário nos poupa minutos preciosos de análise."

Rafael Santana Oliveira, mestre em Ciência da Computação pela UFPA.

"Comentários bons são os que explicam a motivação, coisas que não estão no código — o comentário em si não é o vilão, mas o problema é quando ele serve como forma de tentar amenizar um código que não está claro (em vez de refatorá-lo)."

Eduardo Guerra, doutor e mestre em Engenharia Eletrônica e Computação pelo ITA.

E o comentário final (desculpem o trocadilho) é: comentário não deve servir para explicar código ruim. Somente o que um código bem limpo não explica por si só pode ter uma ajudinha de comentários.

No próximo capítulo, veremos que, em um código limpo, a forma é tão importante quanto o conteúdo.

Capítulo 5
Formatação de código

Depois de aprender sobre várias técnicas de Código Limpo, como nomes de entidades, funções e comentários de código, pode ser que você esteja se sentindo cada vez mais confiante com relação à qualidade de sua programação. De fato, as técnicas abordadas até aqui já dão um considerável *up* na qualidade de um código, deixando-o bem mais limpo.

Quanto ao João Kacau, certo dia ele estava lembrando como eram seus textos na época da escola. Ele pensou: "Eu sempre passava de ano, pois sempre dava as respostas corretas". Então lembrou que nunca tirava nota máxima. O motivo: sua letra não era das melhores e suas respostas, embora com conteúdo correto, eram visualmente bagunçadas, com texto desalinhado, letras de tamanhos diferentes etc.

Algo parecido pode ocorrer com nossos códigos: podem estar funcionando perfeitamente, porém com sérios problemas de formatação. A seguir, veremos as técnicas de Código Limpo para que seu código seja, além de funcional e sem bugs, bem formatado.

Em minha carreira de mais de 10 anos como professor, lembro-me de provas que corrigi, às quais tive vontade de dar uma nota maior do que a resposta merecia só pela beleza da letra e/ou organização do texto. Assim como já tive vontade de dar uma nota menor do que a merecida quando queimei muitos neurônios para decifrar letras terríveis e/ou textos muito mal organizados.

Vamos pensar em comida agora: você já percebeu o quanto achamos que uma refeição ou bolo ou doce é gostoso só porque está com uma bela aparência? Chegamos a achar até que foi feito

de maneira mais higiênica só pela aparência e/ou pela forma como está embalado, não é? Da mesma forma deve ser o nosso código: deve ser bonito — lindo, se possível.

Mas as IDEs formatam nosso código com um simples atalho. Por que eu deveria me preocupar com isso?

De fato, as IDEs atuais fazem um trabalho muito bom na formatação de código. Porém, esses assistentes possuem limites, já que basicamente indentam o código, incluem uma linha em branco entre funções que estão "coladas" e promovem a quebra de linhas para evitar textos longos demais na horizontal. Muitas vezes eles não aplicam certos tipos de formatação que só um programador organizado é capaz de aplicar. E é principalmente nesses tipos de formatação que focaremos neste capítulo.

Em Python, a indentação de blocos de código é obrigatória. Então o código já é formatado por conta disso?

Não. Formatação vai além de indentar o código em funções, classes e estruturas de controle e decisão. Envolve outras técnicas, conforme será apresentado a seguir.

Opinião profissional

"A formatação de código facilita a leitura, e quanto mais pudermos facilitar a leitura de um código, melhor. Considero imperdoável não formatar o código."

Rodrigo Vieira Pinto, mestre em Engenharia de Software pelo IPT-SP.

5.1 Indentação

Embora as IDEs modernas façam um grande trabalho na indentação automática, algumas indentações manuais são necessárias para um código limpo. Caso você não tenha certeza do conceito de **indentação**, podemos dizer que uma linha de código está indentada quando ela inicia com espaços em branco ou tabulações. Os códigos a seguir, por exemplo, possuem vários níveis de indentação.

Java:

```
class Cofrinho {

    void depositar(valor) {
        this.depositos++;
        this.valorDepositado += valor;
    }
}
```

Python:

```
class Cofrinho():

    def depositar(self, valor):
        self._depositos += 1
        self._valor_depositado += valor
```

A primeira linha de ambos os códigos não possui nenhuma indentação. A linha da assinatura do método `depositar` está indentada. Todas as linhas do corpo desse método estão mais indentadas do que a linha da assinatura — podemos dizer que o corpo do método está com *maior nível de indentação*.

Na maioria das linguagens, como Java, Kotlin, Groovy, C# e JavaScript, a indentação é opcional. Porém, é necessário usar a indentação como ferramenta de formatação e organização quando uma instrução ficou grande demais para uma linha e precisamos "quebrar" a linha em duas ou mais. Vejamos o exemplo de código Java a seguir.

```
Cofrinho meuCofre = Cofrinho.builder().dono("Zé
Ruela").valorInicial(1000.0).meta(9000.0).descricaoMeta("Viagem ao
Caribe");
}
```

Não se preocupe com esse `builder()` e os demais métodos invocados. Apenas considere que é uma linha de código válida onde instanciamos um objeto do tipo `Cofrinho` informando quatro atributos por meio de quatro métodos em vez de um construtor. A linha ficou um tanto grande, não acha? É comum usarmos a quebra de linhas e indentação para formatarmos nosso código e facilitar sua leitura e compreensão. Veja como o código ficaria melhor na versão a seguir.

```
Cofrinho meuCofre = Cofrinho.builder()
                            .dono("Zé Ruela")
                            .valorInicial(1000.0)
                            .meta(9000.0)
                            .descricaoMeta("Viagem ao Caribe");
}
```

Perceba como há um esforço mental menor para ver quantos e quais foram os atributos usados para criar um novo `Cofrinho`.

Outro detalhe é o nível de indentação usado: foi o suficiente para deixar fácil de perceber que usamos os métodos de atribuição de valor a partir do `builder()` do `Cofrinho`. Uma indentação automática de uma IDE muitas vezes não usa um nível de indentação suficiente, deixando o código um pouco confuso.

É comum um formatador automático de IDE deixar o código como no exemplo a seguir.

```
Cofrinho meuCofre = Cofrinho.builder()
    .dono("Zé Ruela")
    .valorInicial(1000.0)
    .meta(9000.0)
    .descricaoMeta("Viajem ao Caribe");
}
```

Note como além de parecer, em um primeiro momento, que `dono()`, `valorInicial()` e os demais surgiram do nada, não existe uma "dica" de que eles vêm do `Cofrinho.builder()`.

Em Python, a indentação é **obrigatória** para as seguintes situações:

- Métodos;
- Corpo de uma função;
- Estruturas de controle e decisão.

O código Python a seguir, por exemplo, seria inválido.

```
def depositar_zuada(self, valor):
self._depositos += 1
self._valor_depositado += valor
```

Essa função `depositar_zuada` provocaria um erro de compilação no Python, com a mensagem *`Expected indented block`*, uma vez que o corpo da função não está indentado.

5.2 Agrupamento de código em linhas

Assim como organizar roupas em gavetas torna mais fácil encontrar as roupas que queremos, agrupar trechos do código-fonte em linhas com espaçamentos (linhas em branco) entre eles deixa nosso código mais organizado. Vejamos uma função já vista em capítulos anteriores, que calcula o vale-transporte (VT).

Java:

```
Double getValorVT(Integer passagensCasaTrabalho, Double salario) {
    if (salario < 1000) {
        return null;
    }
    var totalPassagensMes = passagensCasaTrabalho * 2 * 22;
```

```
    var vtMes = totalPassagensMes * 4.5;
    var valorLimite = salario * 0.06;
    if (vtMes > valorLimite) {
        return valorLimite;
    } else {
        return vtMes;
    }
}
```

Python:

```
def get_valor_VT(passagens_casa_trabalho, salario):
    if (salario < 1000):
        return None
    total_passagens_mes = passagens_casa_trabalho * 2 * 22
    vt_mes = total_passagens_mes * 4.5
    valor_limite = salario * 0.06
    if (vt_mes > valor_limite):
        return valor_limite
    else:
        return vt_mes
```

Propositalmente o código está sem nenhuma linha em branco. Percebe como fica um pouco difícil a leitura? Vejamos como ficaria o código agrupado por blocos de linhas.

Java:

```
Double getValorVT(Integer passagensCasaTrabalho, Double salario) {

    if (salario < 1000) {
        return null;
    }

    var totalPassagensMes = passagensCasaTrabalho * 2 * 22;
    var vtMes = totalPassagensMes * 4.5;
    var valorLimite = salario * 0.06;

    if (vtMes > valorLimite) {
        return valorLimite;
```

```
    } else {
        return vtMes;
    }

}
```

Python:

```
def get_valor_VT(passagens_casa_trabalho, salario):

    if (salario < 1000):
        return None

    total_passagens_mes = passagens_casa_trabalho * 2 * 22
    vt_mes = total_passagens_mes * 4.5
    valor_limite = salario * 0.06

    if (vt_mes > valor_limite):
        return valor_limite
    else:
        return vt_mes
```

Percebeu que algumas linhas em branco deixaram o código mais fácil e confortável de ler e entender? Isolamos as estruturas de controle (os `if`) e as linhas com cálculos, bem como demos um espaço em branco entre a assinatura e o corpo da função. Um atalho de formatação de código de IDE dificilmente formataria o código dessa forma. Assim como cartas, artigos e letras de músicas ficam bem melhor agrupados em parágrafos, precisamos agrupar nosso código.

Porém, da mesma forma como parágrafos pequenos demais também não são bons, devemos evitar o excesso de linhas em branco. Vejamos o código a seguir:

Java:

```
String nome = "A definir";


Double saldo = 0.0;
```

```
Integer depositos = 0;
```

Python:

```
nome = "A definir"

saldo = 0.0

depositos = 0
```

Houve apenas uma natureza de instrução: criação de variáveis (ou poderiam ser atributos de instância). Então uma linha em branco entre as instruções acabou aumentando o código na vertical desnecessariamente. O código poderia, portanto, ficar como o a seguir.

Java:

```
String nome = "A definir";
Double saldo = 0.0;
Integer depositos = 0;
```

Python:

```
nome = "A definir"
saldo = 0.0
depositos = 0
```

5.3 Olha que linha grande! Cuidado!

Se você usa IDEs como Eclipse, IntelliJ ou NetBeans, talvez já tenha notado que há uma linha vertical se rolar a barra de rolagem na horizontal ou mesmo se estiver em uma linha de código bem grande (com mais de 150 caracteres, por exemplo).

Linhas de código grandes não costumam causar problemas de compilação, muito menos erros em tempo de execução. Porém, recomenda-se que uma linha de código não seja grande demais. Os principais motivos são:

- Em uma IDE, um código longo na horizontal obriga um leitor a usar a rolagem horizontal, o que não é nada ergonômico.
- Às vezes, é necessário ler um código-fonte a partir de um leitor de texto simples, de uma página de sistema de controle de versão (GitHub, GitLab etc.) ou de um arquivo de log. Nesses casos, um texto longo demais é de difícil leitura.

Na obra original sobre Código Limpo, Robert C. Martin afirma que respeita um máximo de **120 caracteres por linha**. Algumas linguagens de programação possuem orientações sobre isso. Vejamos as orientações para Java e Python:

- **Java**: segundo o *Java Code Conventions*, um documento da década de 1990, a orientação é que uma linha não tenha mais de **80** caracteres. Porém, o *Google Java Style Guide*, que é mais recente, de 2013, e recebe atualizações constantes, sugere um máximo de **100** caracteres por linha.

- **Python**: O *PEP 8 - Style Guide for Python Code*, criado em 2001 e atualizado pela última vez em 2013, orienta que uma linha de código não passe de **79** caracteres, porém afirma que é possível usar um limite de até **100**.

Neste capítulo, vimos algumas técnicas para deixar o código mais formatado e, portanto, mais organizado. Lembre-se de que seu código pode ser (e muito provavelmente será) lido por outra pessoa, que pode ser alguém bem iniciante na programação — ou até mesmo você, depois de anos. Por isso, a organização deve facilitar a vida desses(as) leitores/programadores(as).

No próximo capítulo, serão apresentadas técnicas de abstração de estruturas de dados de objetos.

Capítulo 6
Objetos

Desde criança, João Kacau sempre foi muito curioso. Sempre quis saber como as coisas funcionam. Quando aprendeu, por exemplo, como funciona um interruptor, desses que usamos para ligar e desligar uma lâmpada, pensou: "Nossa, ainda bem que não manipulamos fios de energia elétrica diretamente! Que perigoso e trabalhoso seria!". Pensou também como também seria perigoso e trabalhoso se não existissem aparelhos ou mecanismos como o sistema de partida do motor de um carro, liquidificador, forno de micro-ondas, pilhas e baterias.

Quando não usamos um nível de abstração suficiente em nosso código, é como se estivéssemos oferecendo um código trabalhoso e perigoso de ser usado. Ou seja, em vez de oferecermos um interruptor para ligar e desligar uma lâmpada, é como se estivéssemos oferecendo fios condutores de energia descascados, fita isolante, uma faca e uma escadinha.

Neste capítulo, mais focado em Programação Orientada a Objetos, serão apresentadas técnicas para aumentar a abstração de código em objetos, deixando-os mais seguros e fáceis de usar.

6.1 Abstração de dados

Na Programação Orientada a Objetos, os atributos de instância são os que representam os dados de uma classe. Você acha que eles deveriam ser privados ou públicos? Para iniciantes em programação, pode parecer que é bem mais prático e simples que eles sejam

públicos. Afinal, para que criá-los como privados e então criar métodos públicos de acesso a eles? Só porque é assim que é cobrado em provas de certificação e/ou porque é assim que o professor cobra na prova?

Na verdade, a questão mais importante não é que atributos de instância devem ou não ser privados. Devemos pensar que eles devem ser abstraídos. Isso aumenta a segurança e a facilidade de sua manipulação. Para entender melhor o que seria abstrair dados/atributos, vejamos a seguir exemplos de uma classe não abstraída.

Java:

```java
public class ContaBancaria {

    public String nomeTitular;
    public double saldo = 0.0;

}
```

Python:

```python
class ContaBancaria :

    def __init__(self):
        self.nome_titular = None
        self.saldo = 0.0
```

A classe `ContaBancaria` possui atributos públicos. Isso parece ser superprático em um primeiro instante, afinal, para obtê-los ou alterá-los, basta fazer algo como os códigos a seguir:

Java:

```java
var conta = new ContaBancaria();

conta.nomeTitular = "João Kacau";
```

```
conta.saldo = 999999;

System.out.println("O titular da conta é " + conta.nomeTitular + " e o saldo é de R$" + conta.saldo);
```

Python:

```
conta = ContaBancaria()

conta.nome_titular = "João Kacau"
conta.saldo = 999999

print(f'O titular da conta é {conta.nome_titular} e o saldo é de R${conta.saldo}')
```

Uau! Que prático, certo? Será? Imagine que você fez dezenas de leituras e alterações dos atributos da classe em várias outras classes no seu projeto. Então surge a necessidade de implementar uma regra na alteração do nome do titular da conta, de tal forma que só aceite valores com pelo menos 5 caracteres. Como você implementaria isso? Faria uma cópia de um `if` em dezenas de lugares?

É aqui que a abstração teria facilitado tudo, pois uma vez que o modo como atualizamos o nome do titular é abstraído em um método público, o que mudamos nesse método passa a valer para todas as dezenas de lugares onde solicitamos a alteração de nome de titular na `ContaBancaria`.

Vejamos uma proposta de como implementar essa abstração.

Java:

```
public class ContaBancaria {

    public double saldo = 0.0;
    private String nomeTitular;

    public void setNomeTitular(String nome) {
        if (nome.length() >= 5) {
```

```
            this.nome = nome;
        }
    }
}
```

Python:

```
class ContaBancaria :

    def __init__(self):
        self.saldo = None
        self.__nome_titular = None # em python não existem atributos
privados de fato

    def set_nome_titular(self, nome):
        if (len(nome) >= 5):
            self.__nome_titular = nome
```

> **Atributos privados em Python**
>
> Em Python, não existe o modificador de acesso `private` nem atributos privados de fato. Porém, ao iniciar o nome com `__` (dois *underscore* ou *underline* seguidos), indicamos que o acesso deveria ser apenas privado. É apenas uma indicação para a pessoa programadora, mas não algo que realmente impeça o atributo de ser acessado de forma pública.

Apesar do aumento do código na classe `ContaBancaria`, agora abstraímos a alteração do nome do titular deixando o atributo privado e criando um método público de alteração dele. Assim, mesmo que invoquemos esse método dezenas de vezes no projeto, caso a regra altere, basta alterar o código em um lugar, que é no método de alteração de nome de titular. A seguir, exemplos de código de como solicitaríamos a alteração do nome.

Java:

```
var conta = new ContaBancaria();
conta.setNomeTitular("João Kacau");
```

Python:

```
conta = ContaBancaria()
conta.set_nome_titular("João Kacau")
```

Imagine que surge outra situação: como saldo é público, com o tempo, os programadores foram alterando o saldo de maneiras que não deveriam e em ocasiões impróprias. Também se notou que não ficava devidamente registrado *O QUE* motivou a alteração do saldo (se foi um depósito, um Pix, um pagamento de conta etc.). Por fim, percebeu-se que não existia um *log* com os detalhes das operações que foram alterando o saldo. Para resolver essa questão, a abstração do saldo também seria a solução. Vejamos como isso poderia ser feito nos códigos a seguir.

Java:

```
public class ContaBancaria {

    public double saldo = 0.0;
    private String nomeTitular;

    public void setNomeTitular(String nome) {
        if (nome.length() >= 5) {
            this.nome = nome;
        }
    }

    public void depositar(valor) {
        // código complexo para registro de 'log' de depósito
        // regras complexas para aprovar o depósito
        this.saldo += valor;
    }

    public void enviarPix(valor, chavePix) {
        // código complexo para registro de 'log' de envio via Pix
        // regras complexas para aprovar o Pix (saldo, horário, dados da
```

```
chave Pix etc.)
        this.saldo -= valor;
    }
}
```

Python:

```
class ContaBancaria :

    def __init__(self):
        self.__saldo = None
        self.__nome_titular = None

    def set_nome_titular(self, nome):
        if (len(nome) >= 5):
            self.__nome_titular = nome

    def depositar(self, valor):
        # código complexo para registro de 'log' de depósito
        # regras complexas para aprovar o depósito
        self.__saldo += valor

    def enviar_pix(self, valor, chave_pix):
        # código complexo para registro de 'log' de envio via Pix
        # regras complexas para aprovar o Pix (saldo, horário, dados da
chave Pix etc.)
        self.__saldo -= valor
```

Note que saldo não é mais um atributo público. Sua alteração ficou abstraída porque ele é privado e pode ser alterado via depósito ou envio de Pix. Um exemplo de uso dessa abstração está nos próximos códigos.

Java:

```
var conta = new ContaBancaria();
conta.depositar(1500.0); // se não houver erro, o saldo aumentará em 1500
conta.enviarPix(22.0, "pix@cafeteria.net.br"); // se não houver erro, o
saldo diminuirá em 22
```

Python:

```
conta = ContaBancaria()
conta.depositar(1500.0) # se não houver erro, o saldo aumentará em 1500
conta.enviar_pix(22.0, "pix@cafeteria.net.br") # se não houver erro, o
saldo diminuirá em 22
```

Com a abstração do atributo `saldo`, temos formas mais claras e seguras de alterar seu valor. Outros exemplos de métodos poderiam ser para pagamento de conta, recebimento de Pix, receber salário, saque etc.

6.2 Princípio do menor conhecimento (lei de Demeter)

Vamos supor que o projeto de nossa classe `ContaBancaria` agora possui as classes `Cliente`, que contém dados do cliente titular da conta, e `Endereco`, que contém dados de um dos vários possíveis endereços do cliente. Nesse caso, o relacionamento entre as classes poderia ser expresso nos códigos a seguir.

Java:

```
public class Endereco {
    // atributos e métodos
}

public class Cliente {

    private Endereco[] enderecos;

    public Endereco[] getEnderecos() {
        return this.enderecos;
    }

    // demais atributos
```

```
    // demais métodos
}

public class ContaBancaria {

    private Cliente clienteTitular;
    private Cliente clienteDependente;

    public Cliente getClienteTitular() {
        return this.clienteTitular;
    }

    public Cliente getClienteDependente() {
        return this.clienteDependente;
    }

    // demais atributos
    // demais métodos
}
```

Python:

```
class Endereco :
    # atributos e métodos

class Cliente :

    def __init__(self):
        self.__enderecos = []

    def get_enderecos(self):
        return self.__enderecos

    # demais atributos
    # demais métodos

class ContaBancaria :

    def __init__(self):
        self.__cliente_titular = None
```

```
        self.__cliente_dependente = None

    def get_cliente_titular(self):
        return self.__cliente_titular

    def get_cliente_dependente(self):
        return self.__cliente_dependente

    # demais atributos
    # demais métodos
```

Considerando que existe o conceito de *endereço principal* de um cliente, com o código atual, para obter o endereço principal do titular de uma conta, seria necessário fazer como nos códigos a seguir.

Java:

```
var conta = new ContaBancaria();
var enderecoPrincipalTitular = conta.getClienteTitular().getEnderecos()[0]
```

Python:

```
conta = ContaBancaria()
endereco_principal_titular = conta.get_cliente_titular().get_enderecos()
[0]
```

Essa sequência de *getters* finalizando com o acesso a um elemento de um vetor traz uma série de problemas, tais como:

- E se for necessário obter esse endereço principal em dezenas de lugares diferentes do projeto?
- E se, por engano, usarmos o índice errado no vetor de endereços?
- E se, por engano, pegarmos o endereço a partir do vetor de endereços de um eventual cliente dependente, em vez do titular (afinal ambos poderiam ter o `getEnderecos()` / `get_enderecos()`)?

Tal abordagem para obter o endereço principal do cliente titular de uma conta está ferindo o **princípio do menor conhecimento**,

também conhecido como **lei de Demeter**. A noção fundamental desse princípio é que um determinado objeto deve assumir o mínimo possível sobre a estrutura ou propriedades de qualquer outro componente, mesmo seus subcomponentes. Na prática, não é muito diferente da abstração de dados vista neste capítulo.

Para aplicarmos esse princípio na obtenção do endereço principal do cliente titular da conta, o código poderia ficar como nos próximos exemplos.

Java:

```
// sem alterações na classe Endereco

public class Cliente {

    // atributo e métodos que já existiam

    public Endereco getEnderecoPrincipal() {
        return this.enderecos[0];
    }
}

public class ContaBancaria {

    // atributo e métodos que já existiam

    public Endereco getEnderecoPrincipalTitular() {
        return this.clienteTitular.getEnderecoPrincipal();
    }
}
```

Python:

```
# sem alterações na classe Endereco

class Cliente :

    # atributo e métodos que já existiam
```

```
    def get_endereco_principal(self):
        return self.__enderecos[0]

class ContaBancaria :

    # atributo e métodos que já existiam

    def get_endereco_principal_titular(self):
        return self.__cliente.get_endereco_principal()
```

Assim, sempre que quisermos obter o endereço principal a partir de uma instância de `ContaBancaria`, bastaria fazer como nos códigos a seguir.

Java:

```
var conta = new ContaBancaria();
var enderecoPrincipalTitular = conta.getEnderecoPrincipalTitular();
```

Python:

```
conta = ContaBancaria()
endereco_principal_titular = conta.get_endereco_principal_titular()
```

Note que, além de o código ficar mais simples caso desejemos o endereço principal a partir da `ContaBancaria`, enquanto o conceito de endereço principal existir, a mudança de como isso é obtido na `ContaBancaria` e em `Cliente` passa a ser um tipo de alteração bem menos preocupante, uma vez que tudo está abstraído nos métodos de obtenção de endereço principal.

6.3 DTO (Data Transfer Objects) — e seus "primos" VO e POJO/POPO

Vimos como as classes deveriam abstrair suas complexidades por meio de seus métodos. Porém, há classes que não possuem

nenhuma complexidade para abstrair, pois sua finalidade é basicamente lidar com dados. Esse tipo de classe pode ter métodos, mas devem ser todos apenas com a finalidade de atribuir ou recuperar valores de seus atributos; por isso, é comum vermos nela construtores que recebem valores para seus atributos. Chamamos essas classes de **DTO** (**Data Transfer Objects** ("objetos de transferência de dados" em tradução livre).

Um exemplo de DTO poderia ser a classe `Endereco`, cujo objetivo seria apenas levar e trazer dados do endereço de alguém entre diferentes camadas da nossa aplicação. Seu código-fonte poderia ficar como o a seguir.

Java:

```
public class Endereco {

    private String logradouro;
    private String numero;
    private String complemento;
    private String cep;

    public Endereco(String logradouro, String numero, String complemento,
String cep) {
        this.logradouro = logradouro;
        this.numero = numero;
        this.complemento = complemento;
        this.cep = cep;
    }

    // métodos públicos para leitura e alteração dos atributos
}
```

Java, usando **Lombok**:

```
@AllArgsConstructor
@Getter
@Setter
public class Endereco {
```

```
    private String logradouro;
    private String numero;
    private String complemento;
    private String cep;

}
```

> **POR QUE NÃO USAR LOGO A ANOTAÇÃO `@DATA` DO LOMBOK?**
>
> Poderíamos ter usado, com certeza. Mas tenha ciência de que ela não cria apenas *getters* e *setters*. Ela também cria um construtor para todos os atributos que não são `final`, um `toString()`, um `equals()` e um `hashCode()` (estes dois últimos usando os valores de todos os atributos). Logo, use `@Data` com sabedoria, pois esse excesso de coisas que ele faz automaticamente pode trazer efeitos colaterais.

Python:

```
class Endereco :

    def __init__(self, logradouro, numero, complemento, cep):
        self.__logradouro = logradouro
        self.__numero = numero
        self.__complemento = complemento
        self.__cep = cep

    # métodos públicos para leitura e alteração dos atributos
```

Note que a classe basicamente só existe por causa de seus dados, ou seja, de seus atributos. Nada de métodos complexos.

DTO x VO x POJO/POPO x JavaBeans

Existem alguns conceitos que se confundem com DTO, que são **VO**, **POJO/POPO** e **JavaBeans**. A seguir, um resumo de cada um deles e suas diferenças para o DTO.

Value Objects **— VO**

"Objetos de valor", em tradução livre. São objetos que não possuem um identificador (que é um atributo ou conjunto de atributos). Seu valor, ou seja, o conteúdo do objeto é o que o identifica. Exemplos concretos disso são *strings* e números inteiros. Veja que `"amor"` é igual a `"amor"` e `7` é igual a `7`. Não é preciso explicar muito mais que isso. O valor por si só é o que identifica o objeto.

Quando criamos uma classe VO, temos que garantir que uma instância dela será sempre igual a outra caso todos os valores de todos os atributos sejam iguais.

Em Java, dois VOs são iguais quando a comparação entre eles usando o método `.equals()` retorna `true`. Ou seja, todos os atributos de um VO devem ser *equals* aos atributos de outro VO do mesmo tipo. Por isso, em Java, todo VO deve sobrescrever os métodos `equals()` and `hashCode()` (ou podemos usar a anotação `@EqualsAndHashCode` do Lombok sobre a classe).

Em Python, dois VOs são iguais quando a comparação usando `==` entre eles retorna `true`. Ou seja, todos os atributos de um VO devem ser `==` aos atributos de outro VO do mesmo tipo.

A confusão entre DTO e VO se dá pelo fato de que possuem mais destaque em seus conteúdos (atributos) do que em seus comportamentos (métodos). Porém, um VO tem que implementar esse conceito de "igual" comentado nos três parágrafos anteriores. Por fim, podemos dizer que *todo VO é um DTO, mas nem todo DTO é um VO*.

Plain Old Java Object **— POJO**

"Objeto Java antigo simples", em tradução livre. Esse termo foi criado por Martin Fowler, Rebecca Parsons e Josh MacKenzie em 2000. São classes Java que não possuem ligação específica com nenhum *framework*. Independente de quantos atributos e métodos possuam e de quão complexos são seus métodos, podem facilmente ser copiadas e coladas em qualquer outro projeto Java sem a necessidade de nenhuma dependência adicional. É possível ver em sites e livros o termo POJO "traduzido" para Python como **POPO (Plain Old Python Object)**, cujo conceito é exatamente o mesmo do POJO.

A diferença entre um POJO/POPO e um DTO é que o primeiro pode ter métodos mais complexos.

JavaBeans

JavaBean é apenas um padrão. É uma classe Java comum, mas que segue algumas convenções:

1. Todas as propriedades são *privadas* e são alteradas via `set` e recuperadas via `get` ou `is` (mais detalhes no capítulo 2);
2. Deve haver um construtor público padrão, ou seja, sem argumentos;
3. Deve implementar a `java.io.Serializable`.

É apenas um padrão, porém muitas bibliotecas dependem disso. Por exemplo, a família de bibliotecas **Spring** e a biblioteca **Jackson** usam bastante esse padrão em seu funcionamento.

Assim, um DTO, um VO, um POJO, todos podem ser um JavaBean se seguirem seu padrão.

Java Records (Java 14+)

Foi lançado na versão 14 do Java um recurso experimental — que deixou de ser experimental na versão 16 — chamado `records`. É basicamente um recurso que permite criar DTOs do tipo VO com

bem menos código. Para entender os tipos de situações que uma `record` simplifica, vejamos a classe Java `Endereco` como uma VO a seguir:

```
public class Endereco {

    private String logradouro;
    private String numero;
    private String complemento;
    private String cep;

    public Endereco(String logradouro, String numero, String complemento,
String cep) {
        this.logradouro = logradouro;
        this.numero = numero;
        this.complemento = complemento;
        this.cep = cep;
    }

    public String getLogradouro() {
        return logradouro;
    }

    public String getNumero() {
        return numero;
    }

    public String getComplemento() {
        return complemento;
    }

    public String getCep() {
        return cep;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
```

```
        Endereco endereco = (Endereco) o;

        if (!Objects.equals(logradouro, endereco.logradouro)) return
false;
        if (!Objects.equals(numero, endereco.numero)) return false;
        if (!Objects.equals(complemento, endereco.complemento))
            return false;
        return Objects.equals(cep, endereco.cep);
    }

    @Override
    public int hashCode() {
        int result = logradouro != null ? logradouro.hashCode() : 0;
        result = 31 * result + (numero != null ? numero.hashCode() : 0);
        result = 31 * result + (complemento != null ?
complemento.hashCode() : 0);
        result = 31 * result + (cep != null ? cep.hashCode() : 0);
        return result;
    }

}
```

Note que essa classe:

1. Não permite alterar os seus atributos (ou seja, eles são imutáveis);
2. Devido aos atributos serem imutáveis, é necessário que ela os receba no construtor;
3. Só possui métodos públicos de recuperação dos valores dos atributos (no caso, os `getters`);
4. Sobrescreve os métodos `equals()` e `hashCode()` usando os valores de todos os atributos, para que uma instância dela seja considerada igual à outra caso todos os atributos tenham o mesmo valor.

Usando a biblioteca `lombok`, o código já ficaria bem menor.

```java
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.EqualsAndHashCode;

@AllArgsConstructor
@Getter
@EqualsAndHashCode
public class Endereco {

    private String logradouro;
    private String numero;
    private String complemento;
    private String cep;

}
```

Apenas adicionando as anotações `@AllArgsConstructor`, `@Getter` e `@EqualsAndHashCode`, nossa classe teria o mesmo comportamento da anterior, que tinha o código bem maior. Porém, o Lombok é uma biblioteca que não faz parte da API padrão do Java. Se seu projeto usa Java 14 em diante, considere usar classes `record`. Veja a seguir como `Endereco` ficaria se fosse uma `record`.

```java
public record Endereco (
    String logradouro,
    String numero,
    String complemento,
    String cep
) { }
```

A versão compilada da `Endereco` aqui ficaria muito parecida com a anterior, com `Lombok`. A diferença seria que seus métodos de acesso aos atributos não seriam `getters`, como é possível ver na figura a seguir.

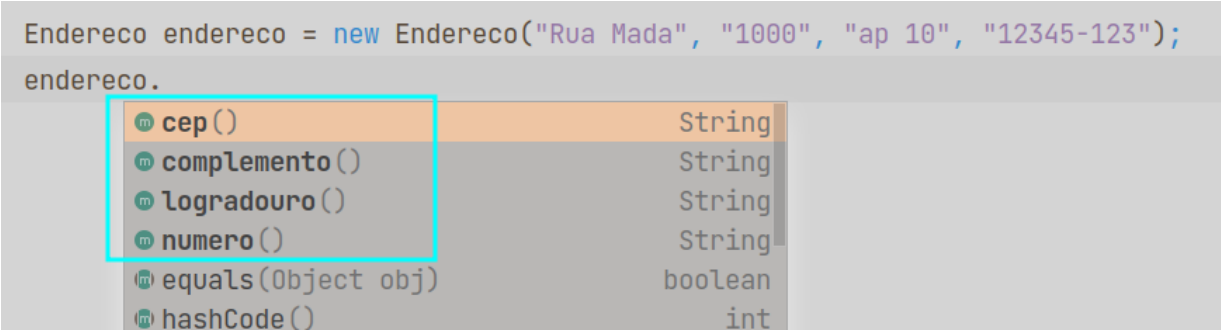


Figura 6.1: Métodos de acesso aos atributos em classes do tipo record.

E, assim como no caso da classe criada com Lombok, os `setters` não existiriam (bem como nenhum outro método de alteração dos valores dos atributos), como é possível ver na próxima figura.

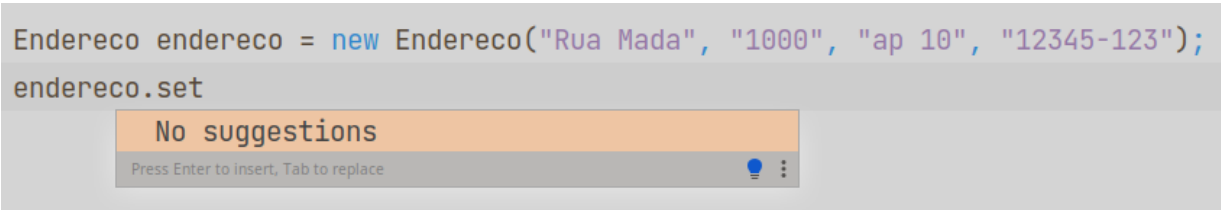


Figura 6.2: Não existem métodos de alteração dos valores de atributos em classes do tipo record.

Se duas instâncias dessa `record` forem criadas com os mesmos valores de atributos, a comparação entre eles com `.equals()` resultaria em `true`.

Por fim, em projetos Java na versão 14 ou mais recente, classes `record` são uma ótima opção para DTOs/VOs. Veja que a `record` acabou tendo características muito parecidas com a classe `Endereco` criada anteriormente, já que, com muito menos código, ela:

1. Não permite que seus atributos sejam alterados;
2. Possui um construtor que recebe todos os seus atributos;
3. Só possui métodos públicos de recuperação dos valores dos atributos;
4. Os métodos `equals()` e `hashCode()` são sobrescritos de maneira transparente e usam os valores de todos os atributos, para que

uma instância dela seja considerada igual à outra caso todos os atributos tenham o mesmo valor.

Esse tipo de classe possui algumas outras possibilidades e restrições, mas falar sobre todas elas exigiria um capítulo inteiro e sairia do escopo desta obra. Se quiser se aprofundar em `record`, a documentação oficial disponível é uma boa referência. Acesse: https://docs.oracle.com/en/java/javase/16/language/records.html.

OPINIÃO PROFISSIONAL

"De modo geral, gosto da ideia de os nossos objetos de domínio terem alguma inteligência. Domínios ricos parecem exigir um pouco mais de maturidade do desenvolvedor, enquanto objetos de domínio anêmicos (estruturas de dados simples) deixam toda a inteligência da aplicação na camada de negócio."

Rafael Santana Oliveira, mestre em Ciência da Computação pela UFPA.

Neste capítulo, vimos algumas técnicas para aumentar a abstração em objetos, tornando seu uso mais fácil e seguro na programação. No próximo capítulo, serão apresentadas técnicas de tratamento de erros.

Capítulo 7
Tratamento de erros

João Kacau foi fazer uma viagem. Chegou cedo ao terminal rodoviário e, enquanto esperava a hora de ir para sua plataforma de embarque, resolveu comprar um chocolate em uma daquelas *vending machines* que têm água, refrigerantes, salgadinhos, chocolates etc. Ele escolheu o que queria, inseriu o cartão de vale-refeição (VR) e digitou a senha.

A máquina então exibiu a mensagem "Transação não efetivada". João pensou ter errado a senha. Fez uma segunda tentativa concentrando-se bem na senha digitada. Porém, novamente tomou a mensagem de "Transação não efetivada". Ele pensou então que o cartão poderia estar sem saldo. Abriu o aplicativo do VR e viu que tinha saldo mais que o suficiente. Em uma última tentativa ele limpou o cartão e... "Transação não efetivada" novamente. Ele já tinha desistido e, quando tirou o cartão, chegou um funcionário do terminal que estava observando a situação. Esse funcionário disse que a máquina não aceita cartão do tipo VR. Foi então que João usou um cartão de débito e... Funcionou!

Um código pode criar um programa que funcione muito bem quando tudo dá certo. Porém, tão importante quanto funcionar é o comportamento de um código **quando algo dá errado**. Assim como devemos saber como lidar com situações adversas do nosso cotidiano, nosso código também deve. Caso contrário, nosso programa pode se tornar um verdadeiro desafio de investigação quando for a hora de lidar com situações que não saíram como o esperado.

Nosso código não deve proporcionar a quem for usá-lo uma experiência como a que João Kacau teve na *vending machine* da rodoviária. Se as técnicas de Código Limpo relacionadas ao tratamento de erros tivessem sido aplicadas, caso o usuário use um

cartão inválido, a mensagem na tela seria algo como *"Este não é um cartão válido"*, fazendo com que a pessoa saiba com clareza qual o problema.

Neste capítulo, serão apresentadas técnicas de implementação de tratamento de erros em um código limpo.

7.1 Diga que houve um erro... E diga qual foi

Imagine que, na situação pela qual João passou na rodoviária, o motivo de seu primeiro cartão não ter sido aprovado é registrado em um arquivo de log ao qual somente o dono da máquina tem acesso. Em que isso ajudaria o usuário da máquina? Em nada, afinal ele não tem acesso a esse arquivo! Da mesma forma, se algo dá errado em uma função que criamos, deve ficar claro qual o problema para quem a usa.

Vejamos como poderia ficar a função `depositar()` (a que definimos no capítulo anterior) com uma nova regra: ela só deve aceitar depósitos com valor acima de 0 (zero):

Java:

```
public void depositar(valor) {

    if (valor <= 0 ) {
        log.error("Só é possível depositar valores acima de 0. Enviado: "
+ valor)
        return;
    }

    this.saldo += valor;
}
```

Python:

```
def depositar(self, valor):

    if (valor <= 0):
        logging.error(f"Só é possível depositar valores acima de 0.
Enviado: {valor}")
        return

    self.__saldo += valor
```

A função está tratando o erro da melhor forma? Em uma primeira análise, pode parecer que sim, afinal, uma mensagem de erro bem clara vai para o log do sistema. Porém, e se um outro código precisar invocar a função `depositar()` ? Esse outro código vai "saber" que houve um problema? Esse outro código vai "saber" se o depósito não foi efetuado em caso de envio de um valor inválido? Vejamos como seria um código que precisa da `depositar()` :

Java:

```
var conta = new ContaBancaria();
Double valorDeposito = // recuperando valor de uma UI, API, banco etc.
conta.depositar(valorDeposito);
// e agora? o depósito aconteceu? como saber?
```

Python:

```
conta = ContaBancaria()
valor_deposito = # recuperando valor de uma UI, API, banco etc.
conta.depositar(valor_deposito);
# e agora? o depósito aconteceu? como saber?
```

Para situações como essas, a orientação para um código limpo seria **lançar uma exceção**. Caso você ainda não tenha muito conhecimento sobre exceções, vale destacar que, quando lançamos uma exceção, a função é interrompida imediatamente e quem a invocou tem como descobrir facilmente qual foi a exceção. Com essa técnica, a função `depositar()` ficaria como segue:

Java:

```
public void depositar(valor) {

    if (valor <= 0 ) {
        var mensagem = "Só é possível depositar valores acima de 0.
Enviado: " + valor;
        log.error(mensagem);
        throw new IllegalArgumentException(mensagem); // se chegar aqui, a
função é interrompida imediatamente
    }

    this.saldo += valor;
}
```

Python:

```
def depositar(self, valor):

    if (valor <= 0):
        mensagem = f"Só é possível depositar valores acima de 0. Enviado:
{valor}"
        logging.error(mensagem)
        raise ValueError(mensagem) # se chegar aqui, a função é
interrompida imediatamente

    self.__saldo += valor
```

Como a função `depositar()` agora lança uma exceção quando a regra de depósitos maiores que `0` for quebrada, quem a invoca pode tratar a exceção de maneira bem mais limpa, bem mais profissional. A seguir, veja como poderia ficar um código que invoca a função `depositar()`, tratando sua possível exceção.

Java:

```
try {
  var conta = new ContaBancaria();
  Double valorDeposito = // recuperando valor de uma UI, API, banco etc.
  conta.depositar(valorDeposito);
} catch (IllegalArgumentException excecao) {
  // se chegar aqui, sabemos que houve problema com o valor enviado à
```

```
'depositar()'
}
```

Python:

```
try:
  conta = ContaBancaria()
  valor_deposito = # recuperando valor de uma UI, API, banco etc.
  conta.depositar(valor_deposito);
except ValueError as excecao:
  # se chegar aqui, sabemos que houve problema com o valor enviado à
'depositar()'
```

Note que agora é bem explícito que pode ocorrer um problema (bloco `try`) e onde ele será tratado caso ocorra (bloco `catch` / `except`).

Breve revisão sobre tratamento de exceções

Uma **exceção** é um erro que ocorre em tempo de execução em um programa, ou seja, quando ele já está em execução em um código sem erros de sintaxe. O conceito de **tratamento de exceções** possui implementações nas mais variadas linguagens de programação modernas. Ele consiste em "avisar" a quem invoca uma função de que algo deu errado e a execução não aconteceu até o fim. As linguagens de programação possuem exceções embutidas (ou "de fábrica"), mas também é possível definir exceções personalizadas, se realmente for necessário.

Exemplos de exceções embutidas em **Java**:

- `IllegalArgumentException`
- `NullPointerException`
- `ArrayIndexOutOfBoundsException`
- `IllegalStateException`
- `ArithmeticException`

Em **Java**, a exceção superclasse de todas as demais é `Throwable`. Os blocos que podemos usar em um código com tratamento de exceção

são `try`, `catch` e `finally`. Se houver um bloco `try`, é obrigatório haver pelo menos um `catch` OU um `finally`. A seguir, veja exemplos de códigos com as estruturas possíveis de tratamento de erro em **Java**:

Bloco `try` com um `catch` e um tipo de exceção

```
try {
  // código que pode lançar uma ou mais exceções
} catch (IllegalArgumentException excecao) {
  // código de tratamento da exceção
}
```

Bloco `try` com um `catch` e vários tipos de exceções

```
try {
  // código que pode lançar uma ou mais exceções
} catch (IllegalArgumentException | ArrayIndexOutOfBoundsException
excecao) {
  // código de tratamento da exceção
}
```

Bloco `try` com vários `catch`

```
try {
  // código que pode lançar uma ou mais exceções
} catch (IllegalArgumentException excecao) {
  // código de tratamento da exceção
} catch (ArrayIndexOutOfBoundsException excecao) {
  // código de tratamento da exceção
}
```

Bloco `try` com `finally`

```
try {
  // código que pode lançar uma ou mais exceções
} finally {
  // código que deve ser executado após o try, mesmo que tenha ocorrido
alguma exceção nele
}
```

Bloco `try` com `catch` e `finally`

```
try {
  // código que pode lançar uma ou mais exceções
} catch (IllegalArgumentException excecao) {
  // código de tratamento da exceção
} finally {
  // código que deve ser executado após o try caso não tenha ocorrido
nenhuma exceção OU após o(s) catch caso alguma exceção ocorra
}
```

Exemplos de exceções embutidas em **Python**:

- `ValueError`
- `TypeError`
- `IndexError`
- `SystemError`
- `ArithmeticError`

Em **Python**, a exceção superclasse de todas as demais é `BaseException`. Os blocos que podemos usar num código com tratamento de exceção são `try`, `except`, `else`, `finally`. Se houver um bloco `try`, é obrigatório haver pelo menos um `except` OU um `finally`. Se for usado o bloco `else`, ele deve estar sempre após o(s) `except` e antes do `finally`. A seguir, veja exemplos de códigos com as estruturas possíveis de tratamento de erro em **Python**:

Bloco `try` com um `except` sem usar o tipo da exceção

```
try:
  # código que pode lançar uma ou mais exceções
except:
  # código de tratamento da exceção
```

Bloco `try` com um `except` e um tipo de exceção

```
try:
  # código que pode lançar uma ou mais exceções
except ValueError as excecao:
  # código de tratamento da exceção
```

Bloco `try` com `except` e `else`

```
try:
  # código que pode lançar uma ou mais exceções
except ValueError as excecao:
  # código de tratamento da exceção
else:
  # código que deve ser executado se não ocorreu nenhuma exceção no try
```

Bloco `try` com um `except` e vários tipos de exceções

```
try:
  # código que pode lançar uma ou mais exceções
except (SystemError, ZeroDivisionError) as excecao:
  # código de tratamento da exceção
```

Bloco `try` com vários `except`

```
try:
  # código que pode lançar uma ou mais exceções
except SystemError as excecao:
  # código de tratamento da exceção
except ZeroDivisionError as excecao:
  # código de tratamento da exceção
```

Bloco `try` com `finally`

```
try:
  # código que pode lançar uma ou mais exceções
finally:
  # código que deve ser executado após o try, mesmo que tenha ocorrido
alguma exceção nele
```

Bloco `try` com `except` e `finally`

```
try:
  # código que pode lançar uma ou mais exceções
except ValueError as excecao:
  # código de tratamento da exceção
finally:
```

```
  # código que deve ser executado após o try caso não tenha ocorrido
nenhuma exceção OU após o(s) except caso alguma exceção ocorra
```

Bloco `try` com `except`, `else` e `finally`

```
try:
  # código que pode lançar uma ou mais exceções
except ValueError as excecao:
  # código de tratamento da exceção
else:
  # código que deve ser executado se não ocorreu nenhuma exceção no try
finally:
  # código que deve ser executado após o try caso não tenha ocorrido
nenhuma exceção OU após o(s) except caso alguma exceção ocorra
```

7.2 Não retorne nulo! Lance uma exceção!

Se invocarmos uma função que foi projetada para retornar um valor, será que é uma boa ideia retornar **nulo** (`null` em Java e `None` em Python)? Vamos usar a função `getEnderecoPrincipal()` / `get_endereco_principal()` do capítulo anterior como exemplo. O código anterior dela considera que sempre há o elemento `0` no vetor de endereços. Mas, e se esse vetor estiver vazio? Em uma primeira refatoração, vamos fazer com que a função retorne **nulo** para a situação de vetor sem elementos (vazio). O código ficaria como segue.

Java:

```
public Endereco getEnderecoPrincipal() {
  return this.enderecos.length > 0 ? this.enderecos[0] : null;
}
```

Python:

```
def get_endereco_principal(self):
  return self.__enderecos[0] if (len(self.__enderecos) > 0) else None
```

No novo código, a função retorna o primeiro elemento do vetor `enderecos` ou **nulo**. Em um primeiro momento pode parecer uma boa ideia, afinal, se não existe um primeiro endereço, retornar **nulo** parece fazer sentido, certo?

Primeiro vamos fazer uma analogia com uma situação do mundo real: você solicita um saque de R$50,00 em um banco 24h. Porém, não há esse valor disponível no equipamento. Ao final da operação, o equipamento diz *"Valor sacado: R$0,00. Retire o valor na saída abaixo"* e essa saída até abre, porém, nada sai dela. Por mais que o valor não seja debitado de sua conta, não seria uma situação estranha? Afinal, se não há o valor desejado no equipamento, ele não deve lhe "dar nada" (R$0,00), mas, sim, avisar que há um problema e o saque não será realizado.

Da mesma forma, se uma função foi projetada para retornar um valor, o fato de ela não ter o que retornar é um problema, é uma situação adversa. Retornar **nulo** é como o caixa do banco 24h que, em vez de avisar que não foi possível realizar o saque, disse que foi sacado R$0,00 e ainda abriu a abertura para retirada de dinheiro nenhum.

Falando agora de consequências na codificação: retornar **nulo** faz com que o código que invoca a função fique sujeito a `NullPointerException` (Java), `AttributeError` ou `TypeError` (Python). No Python, as mensagens de erro são bem mais claras sobre o ponto exato do `AttributeError` / `TypeError` , mas, em Java, achar a real causa de uma `NullPointerException` pode ser um desafio bem estressante.

Por isso, em vez de retornar **nulo**, o melhor seria lançar uma **exceção**. A obtenção do endereço por meio da função `getEnderecoPrincipal()` / `get_endereco_principal()` ficaria como a do código a seguir.

Java:

```
public Endereco getEnderecoPrincipal() {
  if (this.enderecos.length == 0) {
    throw new ArrayIndexOutOfBoundsException("Nenhum endereço foi
definido");
  }

  return this.enderecos[0];
}
```

Python:

```
def get_endereco_principal(self):
  if (len(self.__enderecos) == 0):
    raise IndexError("Nenhum endereço foi definido")

  return self.__enderecos[0]
```

Nessa nova versão do código, quem invocar a função de obtenção de endereço principal vai tomar uma exceção com a mensagem de erro clara caso ainda não exista um endereço principal. Mesmo que quem a invoque não a trate em um bloco `try - catch` / `try - except`, no caso de endereço principal inexistente, o log será bem específico e claro sobre o motivo.

7.3 Crie suas próprias exceções

Linguagens de programação como Java e Python têm um conjunto grande o suficiente para abranger boa parte das situações de exceções que podem ocorrer num sistema. Porém, pode ser necessário ter exceções personalizadas para situações específicas.

Para exemplificar, vamos usar novamente a função `depositar()`. Vamos supor que, além da regra de não permitir que ocorram depósitos menores ou iguais a zero, o valor do saldo não pode ultrapassar 20 mil com o novo depósito.

Uma primeira opção seria usar a mesma exceção (`IllegalArgumentException` / `ValueError`), mudando apenas a mensagem. Porém, imagine que a ação que devemos tomar seja diferente para cada uma das regras. Bom, é possível analisar a mensagem de erro da exceção, mas, será que verificar o conteúdo de uma mensagem de erro é uma boa abordagem?

Talvez usar outra exceção nativa da linguagem de programação para a nova regra seja uma boa opção... Mas será que existe uma classe exceção para cada uma das regras que pensamos para a `depositar()` ? Para situações como essa, uma boa solução pode ser criarmos nossas próprias exceções.

Como exemplo, para a regra que diz que depósitos devem ser maiores que zero, vamos criar a `ValorMinimoException` e, para a regra de que o saldo não pode ultrapassar 20 mil, vamos criar a `LimiteSaldoException` . O código delas poderia ser como o dos próximos códigos.

Java:

```java
// primeira classe
public class ValorMinimoException extends RuntimeException {

    public ValorMinimoException(String mensagemErro) {
      super(mensagemErro);
    }
}

// segunda classe
public class LimiteSaldoException extends RuntimeException {

    public LimiteSaldoException(String mensagemErro) {
      super(mensagemErro);
    }
}
```

Python:

```
# primeira classe
class ValorMinimoException(Exception):

# segunda classe
class LimiteSaldoException(Exception):
```

A função `depositar()` com as novas exceções ficaria como o código a seguir.

Java:

```
public void depositar(valor) {

    if (valor <= 0 ) {
        var mensagem = "Só é possível depositar valores acima de 0.
Enviado: " + valor;
        log.error(mensagem);
        throw new ValorMinimoException(mensagem);
    }

    if ( (valor + this.saldo) > 20_000 ) { // 20_000 é o mesmo que 20000
      throw new LimiteSaldoException("Saldo mais " + valor + " ultrapassa
20 mil");
    }

    this.saldo += valor;
}
```

Python:

```
def depositar(self, valor):

  if (valor <= 0):
    mensagem = f"Só é possível depositar valores acima de 0. Enviado:
{valor}"
    logging.error(mensagem)
    raise ValorMinimoException(mensagem)
```

```
  if ( (valor + self.__saldo) > 20_000 ) # 20_000 é o mesmo que 20000
    raise LimiteSaldoException(f"Saldo mais {valor} ultrapassa 20 mil")

  self.__saldo += valor
```

Agora, veja como ficaria bem claro o tratamento de cada uma das exceções por um código que invoca a `depositar()`:

Java:

```
try {
  var conta = new ContaBancaria();
  Double valorDeposito = // recuperando valor de uma UI, API, banco etc.
  conta.depositar(valorDeposito);

} catch (ValorMinimoException excecao) {
  // se chegar aqui, sabemos que foi enviado um valor menor ou igual a 0

} catch (LimiteSaldoException excecao) {
  // se chegar aqui, sabemos que o valor mais o limite atual ultrapassaria
20 mil
}
```

Python:

```
try:
  conta = ContaBancaria()
  valor_deposito = # recuperando valor de uma UI, API, banco etc.
  conta.depositar(valor_deposito);

except ValorMinimoException as excecao:
  # se chegar aqui, sabemos que foi enviado um valor menor ou igual a 0

except LimiteSaldoException as excecao:
  # se chegar aqui, sabemos que o valor mais o limite atual ultrapassaria
20 mil
```

7.4 Em projetos Java, use exceções não checadas

Aqui temos uma recomendação de Código Limpo bem específica para a linguagem Java, que é a de priorizar o uso de **exceções não checadas**.

Em Java, temos dois tipos de exceções: as **checadas** e as **não checadas**. As exceções *checadas* são aquelas que são subclasses de `Exception`. Quando a lançamos em uma função, sua assinatura deve conter a instrução `throws <Classe de exceção checada>`, e todo código que invocar essa função deverá ou ter um bloco `try-catch` para tratar essa exceção ou ter a mesma instrução `throws <Classe de exceção checada>` em sua assinatura.

Já as exceções **não checadas**, quando lançadas no código de uma função, não provocam nenhuma obrigatoriedade de mudança na assinatura do método nem de tratamento por parte de quem invoca a função. Esse tipo de classe de exceção é subclasse da `RunTimeException` do Java.

A recomendação de usar exceções **não checadas** existe porque elas não obrigam que implementemos código de tratamento de erro em toda uma cadeia de codificação em diferentes classes. Essa propagação de código obrigatória que as exceções **checadas** provocam acaba ferindo o princípio **Open-closed** do **SOLID** (ver tópico sobre SOLID no capítulo sobre Classes).

Não existe esse tipo de orientação para Python porque não existem exceções checadas nela.

Opinião profissional

"A utilização de exceções é necessária quando sabemos que podem ocorrer fluxos de certa forma inesperados. Considero o retorno de nulo como muito pobre, podendo dificultar o entendimento do código."

Renan Rodrigo, mestre em Ciência da Computação pelo IME-USP.

Neste capítulo, vimos a importância de tratar exceções e vários exemplos de como fazê-lo. No próximo capítulo, serão apresentadas técnicas para lidar com limites de códigos externos.

Capítulo 8
Testes de unidade (ou unitários)

João Kacau revolveu aprender a cozinhar e se matriculou em um curso de culinária da famosa Anna Marya Fraga. Ele se esforçou muito nas aulas, anotava tudo e praticava em casa. Certo dia ele convidou vários amigos para jantar na sua casa, e ele faria toda a refeição, pondo em prática o que tinha aprendido no curso. Antes de cozinhar, deixou tudo organizado, fez o chamado *Mise en place*. Durante a preparação, seguiu as receitas e fez tudo com a maior higiene. Não tinha como dar errado, certo?

Ele pediu para os amigos serem sinceros, e eis que disseram que o arroz ficou salgado demais e que a textura da sobremesa não estava boa. O que faltou? Faltou João provar a comida enquanto cozinhava! E provar no momento certo e do jeito certo. Bom, para a sorte do João, a próxima aula do curso de culinária seria sobre técnicas de como provar a comida. Vamos torcer para ele se redimir no próximo jantar que der a seus amigos. ;)

O que aconteceu nessa estorinha pode acontecer no nosso cotidiano de programação caso nosso código não contenha algo chamado **testes de unidade** (ou **testes unitários**), que estudaremos a seguir.

8.1 Por que criar testes de unidade (ou testes unitários)?

Porque software de qualidade sem testes é pura sorte! A palavra **teste** tem uma forte ligação com **qualidade** em qualquer área de trabalho. Exemplos: governos só liberam a venda de medicamentos que foram devidamente *testados*. Você usaria um medicamento

sabendo que não foi testado com rigor? Novos modelos de aviões só começam a ser vendidos após vários *testes*. Você embarcaria em um avião sabendo que não foi suficientemente testado? Portanto, o mesmo vale para software: para a criação de um programa de qualidade, é preciso submetê-lo a vários *testes*.

Mas eu estou usando as técnicas de Código Limpo que aprendi até aqui e meu software terá qualidade, não é? Não necessariamente. As técnicas ensinadas até agora ajudam imensamente na **qualidade do código**, porém podem não garantir a **qualidade do seu funcionamento**. O código pode estar superorganizado e fácil de entender, porém estar simplesmente fazendo a coisa errada: um cálculo errado; devolvendo a mensagem errada; retornando um resultado errado para um cenário específico; dando uma exceção onde não deveria. Por isso, um código deve ser, além de bem escrito, bem testado.

Quando estamos aprendendo a programar, normalmente nossos testes se resumem a executar o programa e validar suas saídas a partir de algumas entradas que fazemos. Isso é um teste? Bom... Sim, mas é o pior tipo de teste. Quando digo isso em aula, alguns alunos dizem que estou sendo "cruel". Estou sendo realista. Sabe por que esse tipo de teste não é bom? Porque quem o executa é um ser humano, portanto:

- A execução do teste depende de uma pessoa disponível e habilitada para realizar o teste;
- A pessoa que testa pode estar cansada, estressada, triste, depressiva, ansiosa etc. Ou seja, a execução do teste pode ser altamente comprometida;
- Quem testa pode não gerar evidência nenhuma de que realizou o teste e como o testou (quando o fez, que valores usou, o que aconteceu ao final), ou pode gerar evidências insuficientes ou de baixa qualidade, seja por imperícia, por má-fé ou por estar como no item anterior (cansado etc.);
- A execução do teste poderá ser imensamente lenta caso executada por uma pessoa que nunca o executou (fora a

chance de execução de forma equivocada).

Como testar bem, então? Usando **testes automatizados**! Testes feitos por um programa, por um robô. Existem vários tipos de testes automatizados, e este capítulo teria centenas de páginas se fôssemos estudar todos eles. No entanto, existe um tipo de teste automatizado que ajuda, e muito, um código a ficar limpo e a garantir sua qualidade de funcionamento: **testes de unidade**, também chamados de **testes unitários** (é que o termo em inglês é ***unit tests***, que admite ambas as formas de tradução).

Testes de unidade são pequenos programas cujo objetivo é testar pequenas partes de um programa. É isso mesmo: é um programa que testa se uma parte do programa está funcionando corretamente.

Quase sempre um teste de unidade testa uma função (ou método) do código, embora existam testes de unidade que validam trechos de configuração ou metaprogramação também. Comparando com a fabricação de um carro, seria como testar, isoladamente, cada pequena peça que compõe o veículo, como o disco de freio ou a buzina, por exemplo.

A seguir, alguns benefícios imediatos que temos pelo fato de os testes de unidade serem executados por um programa:

- Velocidade infinitamente superior na execução dos testes;
- Como possuem um código-fonte por serem também programas, servem como evidência de que foram feitos testes;
- As evidências dos resultados dos testes são geradas automaticamente (*logs*);
- Podem simular situações impossíveis para pessoas, como dezenas, centenas, milhares, milhões de execuções simultâneas;
- Todos os testes programados serão executados. É um programa, logo não ocorre a omissão ou execução incorreta de alguns testes por cansaço, ansiedade, preguiça, má-fé etc.;

- Podem ser reexecutados sempre que necessário. Isso é extremamente útil para descobrir se mudanças em uma parte código-fonte do projeto não tiveram algum impacto negativo inesperado em outra parte.

Outro benefício muito legal dos testes automatizados é que, enquanto os criamos, podemos encontrar problemas de *design* no código testado, como excesso de métodos estáticos, alto acoplamento, baixa coesão (há explicações sobre isso no capítulo sobre Classes) etc. Um código difícil de ser testado é, muito provavelmente, um código que precisa ser melhorado.

Por fim, testes automatizados em um projeto costumam ser executados automaticamente em processos de *build* e integração contínua. Se um teste não passar, o processo é interrompido e o projeto não é levado para o ambiente de produção.

8.2 Qual a tradução mais correta? Testes "de unidade" ou "unitários"?

Até aqui foram apresentados ambos os termos usados no Brasil, "testes de unidade" e "testes unitários", que seriam a tradução de *unit test*, o termo original em inglês. Mas ambos estão corretos? A única bibliografia que encontrei foi um tópico de outubro de 2012 em um fórum on-line, onde um aluno de doutorado em Ciência da Computação da USP — Maurício Aniche — disse ter entrado em contato com a **Academia Brasileira de Letras** (ABL) perguntando sobre esse ponto.

A pergunta do Mauricio foi:

Olá, Meu nome é Mauricio Aniche e sou aluno de doutorado em Ciência da Computação na Universidade de São Paulo. Uma dúvida muito grande em nossa área é sobre o uso da palavra "unitário" ou

da expressão "de unidade". Um programa de computador é composto por diversas unidades que trabalham juntas. Uma das etapas do processo de desenvolvimento é justamente testar cada uma dessas unidades de maneira isolada e garantir que elas funcionem adequadamente. Chamamos essa parte do processo de "teste de unidade". A grande dúvida é: usar a expressão "teste unitário" é errado neste contexto? No dicionário, encontrei que "unitário" significa "relativo à unidade".

E a resposta da **ABL** foi:

Prezado Maurício, termos técnicos de áreas específicas devem ficar a cargo de especialistas da área. Teste unitário, em princípio, é um único teste. Não se trata de certo ou errado. Em tese, se a expressão é usual na área citada, está adequada.

O tópico de fórum citado encontra-se aqui: https://groups.google.com/g/tdd-no-mundo-real/c/yOxP4fEeLcE?pli=1.

Portanto, já que ambas as expressões são usuais, ambas são adequadas. Mas se você é daqueles que gostam de rigor na semântica e derivação das palavras, então pode afirmar que a tradução "testes de unidade" é a mais apropriada. Afinal, esse tipo de teste tem como objetivo testar uma pequena unidade do código. Para evitar citar ambas as traduções o tempo todo neste capítulo, será citada apenas essa de agora em diante.

8.3 Como criar testes de unidade

Qualquer linguagem de programação permite a criação desse tipo de teste, afinal é só criar um programa que testa uma pequena parte de outro. Esse conceito está tão consolidado que as principais linguagens de programação do mundo possuem bibliotecas e/ou *frameworks* para facilitar a criação e execução de testes de unidade.

Em Java, não existe um mecanismo em sua biblioteca padrão que facilite a criação de testes de unidade, porém a biblioteca mais recomendada para isso é o **JUnit**, que, até o momento da escrita desta obra, estava na versão `5.x` (lançada em 2016). Essa será a versão usada nos exemplos deste livro.

Para adicioná-lo em um projeto Java, basta adicionar sua biblioteca por meio de do gerenciador de dependência do projeto (Maven ou Gradle). Alguns frameworks como o **Spring Boot**, por exemplo, dispensam isso, pois já têm a dependência do JUnit neles.

Em Python, temos o **unittest**, que vem em sua biblioteca padrão. Ele facilita a criação e execução de testes unitários. Nesta obra, foi usada a versão `3.8.x`. Uma vez que vem embarcado na biblioteca padrão do Python, não é preciso fazer nenhuma configuração extra para usá-lo.

Vale reforçar que o objetivo deste capítulo não é ser um tutorial de uso de **JUnit** nem **unittest**, o que o faria ter centenas de páginas. A intenção aqui é mostrar brevemente como são feitos testes de unidade com essas ferramentas e demonstrar como eles podem ajudar a melhorar a qualidade do funcionamento de um programa.

8.4 Estudo de caso — Calculadora de vale-transporte (VT)

Para vermos como seriam testes de unidade e como podem ajudar na melhoria da qualidade do funcionamento de um programa, vamos testar uma calculadora de vale-transporte (VT). Ela deveria funcionar assim:

1. Deve haver uma classe `CalculadoraSalario`;
2. Essa classe deve ter um método que recebe três parâmetros: o salário bruto, o valor da passagem e a quantidade de viagens

que a pessoa faz por dia para ir e voltar do trabalho. Ele deve retornar o valor do VT;
3. O cálculo para o VT é:
    - 3.1. Multiplica-se o valor da passagem pela quantidade de viagens que a pessoa faz por dia para ir e voltar do trabalho;
    - 3.2. Se o valor dessa multiplicação for de até 6% do salário bruto, o VT será esse valor;
    - 3.3. Se o valor dessa multiplicação for maior do que 6% do salário bruto, o VT será 6% do salário bruto.
4. Caso qualquer um dos três parâmetros seja um valor negativo, uma exceção deve ser lançada (`IllegalArgumentException` para Java ou `ValueError` para Python).

Os cenários que podemos usar para validar se nosso método de cálculo de VT está correto poderia ser como o da tabela abaixo. Aliás, esse tipo de tabela, com cenários de teste para a validação de um programa, pode ser chamado de **teste de mesa**:

Cenário	Salário	Valor da passagem	Viagens por dia	Resultado
1	5000.0	2.0	2	88.0
2	2000.0	2.0	2	88.0
3	2000.0	5.0	2	120.0
4	-2000.0	2.0	2	*Exceção* (`IllegalArgumentExc` / `ValueError`)
5	2000.0	-2.0	2	*Exceção* (`IllegalArgumentExc` / `ValueError`)
6	2000.0	2.0	-2	*Exceção* (`IllegalArgumentExc`

Cenário	Salário	Valor da passagem	Viagens por dia	Resultado
				/ `ValueError`)

A classe `CalculadoraSalario` com o método de cálculo de VT seria como a dos exemplos a seguir.

Java:

```
public class CalculadoraSalario {

    public Double getDescontoVT(Double salario, Double valorPassagem, int
viagensDia) {
        // implementação do método
    }
}
```

Python:

```
class CalculadoraSalario :
    def  get_desconto_VT(self, salario, valor_passagem, viagens_dia) :
      # implementação do método
```

As classes de teste para a classe e método de cálculo de VT seria como as dos exemplos a seguir.

Java:

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class CalculadoraSalarioTest {

    @Test
    void testGetDescontoVT() {
        CalculadoraSalario calculadora = new CalculadoraSalario();

        Double desconto = calculadora.getDescontoVT(5000.0, 2.0, 2);
        assertEquals(88.0, desconto);
```

```
        desconto = calculadora.getDescontoVT(2000.0, 2.0, 2);
        assertEquals(88.0, desconto);

        desconto = calculadora.getDescontoVT(2000.0, 5.0, 2);
        assertEquals(120.0, desconto);

        assertThrows(IllegalArgumentException.class, () ->
calculadora.getDescontoVT(-2000.0, 2.0, 2));
        assertThrows(IllegalArgumentException.class, () ->
calculadora.getDescontoVT(2000.0, -2.0, 2));
        assertThrows(IllegalArgumentException.class, () ->
calculadora.getDescontoVT(2000.0, 2.0, -2));
    }
}
```

Python:

```
import unittest

from CalculadoraSalario import CalculadoraSalario

class CalculadoraSalarioTest(unittest.TestCase):

    def test_get_desconto_VT(self):
        calculadora = CalculadoraSalario()

        desconto = calculadora.get_desconto_VT(5000.0, 2.0, 2)
        self.assertEqual(88.0, desconto)

        desconto = calculadora.get_desconto_VT(2000.0, 2.0, 2)
        self.assertEqual(88.0, desconto)

        desconto = calculadora.get_desconto_VT(2000.0, 5.0, 2)
        self.assertEqual(120.0, desconto)

        with self.assertRaises(ValueError):
            calculadora.get_desconto_VT(-2000.0, 2.0, 2)

        with self.assertRaises(ValueError):
```

```
            calculadora.get_desconto_VT(2000.0, -2.0, 2)

        with self.assertRaises(ValueError):
            calculadora.get_desconto_VT(2000.0, 2.0, -2)

if __name__ == '__main__':
    unittest.main()
```

Notou que a palavra `assert` estava em vários lugares dos códigos? São os momentos em que são feitas as **asserções**, ou **verificações**. É nas asserções que verificamos se o resultado foi o esperado. Se uma asserção falha, o teste **falha**. Se nenhuma asserção falha, o teste **passa**. Caso ocorra algum erro não previsto na execução do teste, dizemos que o teste resultou em **erro**. Ratificando, o resultado da execução de um teste pode ser:

- **Passou** — Todas as asserções deram certo;
- **Falhou** — Uma ou mais asserções falharam;
- **Erro** — Uma ou mais asserções não puderam sequer ser executadas devido a um erro não tratado durante a execução dos testes.

Note que escrevemos um código, um programa que testa nosso método de cálculo de VT. Assim, temos a evidência física de que **foram feitos testes** e de **como foram feitos**.

Vamos supor que, na implementação do método de cálculo de VT, nós não programamos corretamente as regras 3.2 e 3.3 e, portanto, o cenário 3 do teste de mesa deu errado. Ao executar o teste de unidade, o *log* de execução seria como o dos exemplos a seguir.

Log **de execução com falha no cenário 3 — Java/JUnit**:

```
-------------------------------------------------------------------------------
Test set: cap09.CalculadoraSalarioTest
-------------------------------------------------------------------------------
Tests run: 1, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.051 s
```

```
<<< FAILURE! - in cap09.CalculadoraSalarioTest
testGetDescontoVT  Time elapsed: 0.043 s  <<< FAILURE!
org.opentest4j.AssertionFailedError: expected: <120.0> but was: <220.0>
    at
cap09.CalculadoraSalarioTest.testGetDescontoVT(CalculadoraSalarioTest.java
:20)
```

Log de execução com falha no cenário 3 — Python/unittest:

```
======================================================================
FAIL: test_get_desconto_VT (__main__.CalculadoraSalarioTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "./cap09/CalculadoraSalarioTest.py", line 17, in
test_get_desconto_VT
    self.assertEqual(120.0, desconto)
AssertionError: 120.0 != 220.0

----------------------------------------------------------------------
Ran 1 test in 0.000s

FAILED (failures=1)
```

Observe como os *logs* de execução descrevem o que era esperado (`120.0`) e o que o método retornou (`220.0`). É indicada até a linha da classe de teste que teve a verificação que falhou.

Agora, vamos supor que corrigimos esse bug. Mas ainda há um problema: nós não programamos corretamente a regra 4 e, portanto, os cenários 4, 5 e 6 do teste de mesa deram errado. Ao executar o teste de unidade, o *log* de execução seria como o dos exemplos a seguir.

Log de execução com falha no cenário 4 — Java/JUnit:

```
-------------------------------------------------------------------------
-----
Test set: cap09.CalculadoraSalarioTest
-------------------------------------------------------------------------
-----
```

```
Tests run: 1, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.158 s
<<< FAILURE! - in cap09.CalculadoraSalarioTest
testGetDescontoVT  Time elapsed: 0.146 s  <<< FAILURE!
org.opentest4j.AssertionFailedError: Expected
java.lang.IllegalArgumentException to be thrown, but nothing was thrown.
    at
cap09.CalculadoraSalarioTest.testGetDescontoVT(CalculadoraSalarioTest.java
:22)
```

Log de execução com falha no cenário 4 — Python/unittest:

```
======================================================================
FAIL: test_get_desconto_VT (__main__.CalculadoraSalarioTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "./cap09/CalculadoraSalarioTest.py", line 19, in
test_get_desconto_VT
    with self.assertRaises(ValueError):
AssertionError: ValueError not raised

----------------------------------------------------------------------
Ran 1 test in 0.000s

FAILED (failures=1)
```

Percebeu como os *logs* de execução dizem que era esperada uma exceção (`IllegalArgumentException` / `ValueError`), que não ocorreu? É indicada até a linha da classe de teste que teve a verificação que falhou.

Vamos supor que corrigimos mais esse bug. Se nosso método de cálculo de VT agora estiver perfeito em termos de funcionalidade, ao executarmos o teste de unidade, o *log* de execução seria como o dos exemplos a seguir.

Log de execução sem falhas — Java/JUnit:

```
-------------------------------------------------------------------------
-----
Test set: cap09.CalculadoraSalarioTest
-------------------------------------------------------------------------
-----
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.157 s -
in cap09.CalculadoraSalarioTest
```

Log de execução sem falhas — Python/unittest:

```
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Observe como os *logs* de erro dizem que não houve nenhuma falha na execução dos testes. Ou seja, todos os cenários passaram! Nesse caso, a pessoa programadora poderia dizer, aliviada, que o método de cálculo de VT está funcionando como deveria!

Vale reforçar que foi criado um programa que testa nosso método em seis cenários. Sempre que necessário, ele pode ser executado com alguns cliques ou teclas de atalho e, em poucos segundos, nos gerará evidências da execução dos testes (os *logs*). Ainda, sistemas de *build* e integração contínua os executariam antes de levar o projeto para o ambiente de produção. Se você ainda não sabe o que são sistema de *build* ou integração contínua, não se preocupe. Estou apenas comentando como os testes unitários são consolidados a ponto de serem executados automaticamente por vários tipos de ferramentas.

Por fim, além da evidência de que foram feitos testes e de como eles foram feitos, após sua execução são geradas evidências do **resultado da execução** desses testes — os *logs*.

Se estou escrevendo testes automatizados, estou adotando o TDD?

Não. Apenas ter testes de unidade não significa que automaticamente estamos usando **TDD** (***Test-Driven Development***, ou *desenvolvimento guiado a testes*, em tradução livre). O TDD é uma prática de desenvolvimento que é conhecida por esse nome desde a década de 1990, graças a Kent Beck, que a batizou e propôs que essa técnica fosse parte da metodologia de desenvolvimento **XP** (***Extreme Programming***). Há, no entanto, registros bibliográficos que mencionam técnicas muito parecidas nas décadas de 1950, 1960 e 1970.

No TDD, o ciclo proposto é:

1. Escrevemos os testes para o código que queremos testar *antes* de implementá-lo;
2. Os testes vão falhar;
3. Escrevemos o código de produção até que os testes passem;
4. Os testes vão passar;
5. Passados os testes, refatoramos o código de produção;
6. O ciclo reinicia.

Figura 8.1: Ciclo TDD.

Se você está criando código de teste para código de produção que já existe, já não está praticando o **TDD**. E isso está longe de ser ruim! Ter testes automatizados no seu código, independentemente de quando foram escritos, eleva muito o patamar de qualidade de seus projetos. Aliás, para iniciantes em programação, criar testes de unidade antes do código de produção pode ser uma missão muito complicada. Porém, escrever um código de teste para uma função que já existe é bem mais simples.

8.5 Boas práticas em testes de unidade

As orientações de Código Limpo sugerem três boas práticas em testes de unidade:

- Uma asserção por teste;
- Um conceito por teste;
- FIRST.

Elas estão descritas a seguir.

Uma asserção por teste

Existe uma corrente de pensamento que defende que cada teste deve ter apenas uma asserção, contemplando somente um cenário de teste. Seria como aplicar o princípio **S** do **SOLID** em funções de teste. Se fôssemos aplicar essa prática, nosso teste seria dividido em seis (um teste por cenário). Vejamos com isso ficaria nos próximos códigos, nos testes dos cenários 1, 3 e 4.

Java:

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class CalculadoraSalarioTest {

    @Test
    void testGetDescontoVTCenario1() {
        CalculadoraSalario calculadora = new CalculadoraSalario();

        Double desconto = calculadora.getDescontoVT(5000.0, 2.0, 2);
        assertEquals(88.0, desconto);
    }

    @Test
    void testGetDescontoVTCenario3() {
        CalculadoraSalario calculadora = new CalculadoraSalario();

        Double desconto = calculadora.getDescontoVT(2000.0, 5.0, 2);
        assertEquals(120.0, desconto);
    }

    @Test
```

```
    void testGetDescontoVTCenario4() {
        CalculadoraSalario calculadora = new CalculadoraSalario();

        assertThrows(IllegalArgumentException.class, () ->
calculadora.getDescontoVT(-2000.0, 2.0, 2));
    }
}
```

Python:

```
import unittest

from CalculadoraSalario import CalculadoraSalario

class CalculadoraSalarioTest(unittest.TestCase):

    def test_get_desconto_VT_cenario1(self):
        calculadora = CalculadoraSalario()

        desconto = calculadora.get_desconto_VT(5000.0, 2.0, 2)
        self.assertEqual(88.0, desconto)


    def test_get_desconto_VT_cenario3(self):
        calculadora = CalculadoraSalario()

        desconto = calculadora.get_desconto_VT(2000.0, 5.0, 2)
        self.assertEqual(120.0, desconto)


    def test_get_desconto_VT_cenario4(self):
        calculadora = CalculadoraSalario()

        with self.assertRaises(ValueError):
            calculadora.get_desconto_VT(-2000.0, 2.0, 2)

if __name__ == '__main__':
    unittest.main()
```

A abordagem dessa boa prática pode não agradar a todas as pessoas que programam porque há funções que podem ter muitos cenários possíveis, o que levaria à criação de arquivos de teste imensos. Há uma outra proposta que não deixaria os testes tão grandes. Vejamos a seguir.

Um conceito por teste

Outra corrente de pensamento defende que cada teste deve ter apenas um conceito, contemplando quantos cenários de teste forem necessários. Por exemplo, poderíamos dividir os testes aqui descritos em dois conceitos: testes com valores válidos e testes com valores inválidos. Vejamos como isso ficaria nos próximos códigos.

Java:

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class CalculadoraSalarioTest {

    @Test
    void testGetDescontoVTValoresValidos() {
        CalculadoraSalario calculadora = new CalculadoraSalario();

        Double desconto = calculadora.getDescontoVT(5000.0, 2.0, 2);
        assertEquals(88.0, desconto);

        desconto = calculadora.getDescontoVT(2000.0, 2.0, 2);
        assertEquals(88.0, desconto);

        desconto = calculadora.getDescontoVT(2000.0, 5.0, 2);
        assertEquals(120.0, desconto);
    }

    @Test
    void testGetDescontoVTValoresInvalidos() {
        CalculadoraSalario calculadora = new CalculadoraSalario();
```

```
        assertThrows(IllegalArgumentException.class, () ->
calculadora.getDescontoVT(-2000.0, 2.0, 2));
        assertThrows(IllegalArgumentException.class, () ->
calculadora.getDescontoVT(2000.0, -2.0, 2));
        assertThrows(IllegalArgumentException.class, () ->
calculadora.getDescontoVT(2000.0, 2.0, -2));
    }
}
```

Python:

```
import unittest

from CalculadoraSalario import CalculadoraSalario

class CalculadoraSalarioTest(unittest.TestCase):

     def test_get_desconto_VT_valores_validos(self):
        calculadora = CalculadoraSalario()

        desconto = calculadora.get_desconto_VT(5000.0, 2.0, 2)
        self.assertEqual(88.0, desconto)

        desconto = calculadora.get_desconto_VT(2000.0, 2.0, 2)
        self.assertEqual(88.0, desconto)

        desconto = calculadora.get_desconto_VT(2000.0, 5.0, 2)
        self.assertEqual(120.0, desconto)


     def test_get_desconto_VT_valores_invalidos(self):
        calculadora = CalculadoraSalario()

        with self.assertRaises(ValueError):
            calculadora.get_desconto_VT(-2000.0, 2.0, 2)

        with self.assertRaises(ValueError):
            calculadora.get_desconto_VT(2000.0, -2.0, 2)

        with self.assertRaises(ValueError):
```

```
            calculadora.get_desconto_VT(2000.0, 2.0, -2)

if __name__ == '__main__':
    unittest.main()
```

Percebeu que aqui agrupamos tipos de cenários de teste em um mesmo código de teste? Assim, diferentes cenários que possuem um mesmo conceito, um mesmo "perfil" de teste, ficam agrupados em uma mesma função de teste. Isso leva a uma redução dos arquivos de testes sem comprometer sua organização.

Não existe melhor ou pior prática entre essa (um conceito por teste) e a anterior (uma asserção por teste). São apenas propostas diferentes de agrupamento de testes.

FIRST

Este acrônimo representa cinco regras para a criação de testes limpos. São elas:

Fast (*Rápido*): os testes devem ser executados muito rapidamente. Por quê? Porque se forem de execução lenta, criam a tentação de serem jogados para escanteio pela equipe de desenvolvimento. Se necessário, os testes devem ser refatorados e bibliotecas de teste, adicionadas ou trocadas.

Independent (*Independente*): os testes devem ser independentes uns dos outros. Pode haver rotinas de preparação compartilhadas por vários testes em um mesmo arquivo, mas um teste não deveria nunca depender de algo que outro teste testou.

Repeatable (*Repetível*): testes bem feitos podem ser reexecutados em vários ambientes diferentes (desenvolvimento, ambiente de QA, homologação etc.). Ou seja, não devem estar atrelados a uma infraestrutura específica.

Self-Validating (*Autovalidação*): o resultado do teste deve ser claro, objetivo e sem ambiguidades. Seu final deve dizer por si só se

ele passou ou falhou, sem a necessidade de uma validação manual complementar.

Timely (*Oportuno*): essa regra é bem específica ao TDD, segundo a qual devemos criar os testes automatizados antes do código que será testado — por exemplo, criaríamos o teste de um método `somar()` antes de criar o corpo desse método, bastando que exista sua assinatura (afinal, sem a assinatura, sequer é possível criar o teste). Essa regra parte do princípio de que, se formos criar os testes só depois de codificar o código de produção, corremos o risco de criar um código de qualidade ruim e, portanto, difícil de ser testado. Esse princípio é uma das bases do TDD e é bem controverso no mundo de desenvolvimento de software.

Pelo menos as quatro primeiras regras devem ser implementadas sempre que possível. A quinta regra, a *timely*, só se for adotar o TDD.

8.6 Cobertura de testes

Até aqui, falamos sobre a importância de criar testes de unidade automatizados, sobre algumas ferramentas que permitem implementá-los e algumas boas práticas. Mas como saber se estamos testando o suficiente em um projeto?

A unidade de medida da quantidade de código que está sendo testado de forma automatizada em um projeto chama-se **cobertura de testes**, ou simplesmente **cobertura**. Ela é sempre expressa em **porcentagem**. Ou seja, se absolutamente todo o código de um projeto é testado por testes de unidade, podemos dizer que tem *100% de cobertura de testes*, ou, ainda, que está *100% coberto*. Já se metade do código de um projeto é testada por testes de unidade, seriam *50% de cobertura de testes*, ou, ainda, que está *50% coberto*.

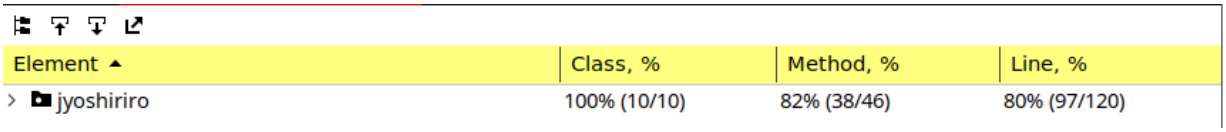
Como calcular a cobertura?

A cobertura de testes em um projeto é sempre expressa em porcentagem, mas pode ter diferentes resultados, dependendo de que características foram usadas para medir essa porcentagem. Em projetos que usam o paradigma de Programação Orientada a Objetos, por exemplo, as coberturas de testes podem ser:

- Cobertura de **Classes**: quantas classes foram usadas em pelo menos um teste;
- Cobertura de **Métodos**: quantos métodos foram invocados por pelo menos um teste;
- Cobertura de **Linhas**: quantas linhas do código-fonte foram testadas.

Como medir a cobertura?

Para medir a cobertura, usamos **ferramentas de análise de cobertura de código**. Algumas funcionam diretamente em IDEs e outras são executadas em *pipelines* de ferramentas de CI/CD. Na figura a seguir, é possível ver um exemplo de relatório de análise de cobertura de código em um projeto. Trata-se de um relatório gerado pela IDE IntelliJ IDEA.



Element	Class, %	Method, %	Line, %
jyoshiriro	100% (10/10)	82% (38/46)	80% (97/120)

Figura 8.2: Relatório de análise de cobertura de código.

A partir desse relatório, podemos afirmar que, no projeto analisado:

- **100% das classes** são usadas por pelo menos um teste;
- **82% dos métodos** foram invocados por pelo menos um teste;
- **80% das linhas** do código-fonte foram testadas.

Qual a melhor medida de cobertura de código?

A medida que dá uma noção mais exata de quanto realmente está testado é a cobertura de **linhas**. É possível perceber isso na próxima imagem, que contém um relatório mais detalhado de análise de cobertura de testes.

Element	Class, %	Method, %	Line, %
jyoshiriro	90% (10/11)	80% (38/47)	80% (97/121)
apiavaliacaofilmes	90% (10/11)	80% (38/47)	80% (97/121)
clientesapi	0% (0/1)	0% (0/1)	0% (0/1)
OmdbClienteApi	100% (0/0)	100% (0/0)	100% (0/0)
OscarApi	0% (0/1)	0% (0/1)	0% (0/1)
config	100% (2/2)	33% (1/3)	19% (4/21)
SegurancaConfig	100% (1/1)	100% (1/1)	100% (3/3)
UsuariosConfig	100% (1/1)	0% (0/2)	5% (1/18)
controle	100% (1/1)	100% (4/4)	100% (8/8)
dominio	100% (2/2)	72% (13/18)	76% (16/21)
AvaliacaoFilme	100% (1/1)	37% (3/8)	54% (6/11)
Partida	100% (1/1)	100% (10/10)	100% (10/10)
repositorio	100% (0/0)	100% (0/0)	100% (0/0)
requisicao	100% (1/1)	100% (6/6)	100% (9/9)
resposta	100% (1/1)	100% (4/4)	100% (4/4)
servico	100% (2/2)	100% (10/10)	100% (55/55)
OmdbService	100% (1/1)	100% (4/4)	100% (14/14)
PartidaService	100% (1/1)	100% (6/6)	100% (41/41)
ApiAvaliacaoFilmesApplication	100% (1/1)	0% (0/1)	50% (1/2)

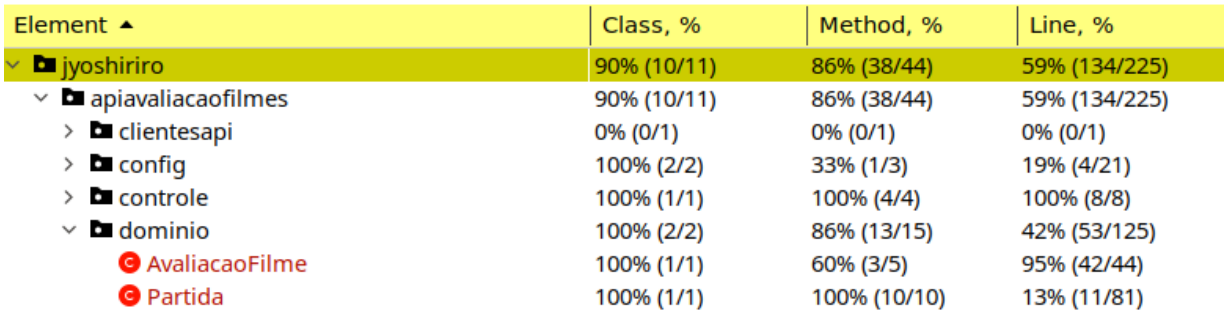
Figura 8.3: Relatório detalhado de análise de cobertura de código.

Note como a cobertura de classes é a menos precisa. Todas as classes no relatório possuem 100% ou 0% de cobertura. Isso porque basta que pelo menos um objeto da classe seja instanciado, ou que pelo menos um atributo ou método estático seu seja usado, para que essa classe passe a ter 100% de cobertura. Ou seja, é uma medida binária: ou é 0% ou 100%.

Já a cobertura de métodos é mais precisa que a de classes, porém ainda não é tão precisa. Isso porque basta que um método seja invocado para ele ter 100% de cobertura, não importando se apenas uma linha dentro dele foi testada. Portanto, para cada método, também temos uma medida binária.

A cobertura de linhas é a mais precisa, por isso é a mais utilizada. Sempre que ela for 100%, as outras medidas (classes e métodos) também serão. E se ela for de 0%, certamente as outras medidas também serão. Ambos os casos estão presentes no relatório.

Para demonstrar por que a cobertura de linhas é mais precisa do que a de métodos, vejamos o próximo relatório de cobertura na figura a seguir.



Element	Class, %	Method, %	Line, %
jyoshiriro	90% (10/11)	86% (38/44)	59% (134/225)
apiavaliacaofilmes	90% (10/11)	86% (38/44)	59% (134/225)
clientesapi	0% (0/1)	0% (0/1)	0% (0/1)
config	100% (2/2)	33% (1/3)	19% (4/21)
controle	100% (1/1)	100% (4/4)	100% (8/8)
dominio	100% (2/2)	86% (13/15)	42% (53/125)
AvaliacaoFilme	100% (1/1)	60% (3/5)	95% (42/44)
Partida	100% (1/1)	100% (10/10)	13% (11/81)

Figura 8.4: Relatório de análise de cobertura de código — métodos x linhas.

Note que a análise da classe `AvaliacaoFilme` diz que ela tem 60% de cobertura de testes nos métodos, mas 95% nas linhas. Ora, 95% (quase 100%) das linhas foram alcançadas por algum dos testes de unidade. Se temos apenas 60% de cobertura nos métodos, certamente temos métodos muito pequenos que faltam serem testados. Logo, faz muito mais sentido apresentar 95% como medida de cobertura de testes dessa classe.

Já a análise da classe `Partida` diz que ela tem 100% de cobertura de testes nos métodos mas 13% nas linhas. Isso mostra novamente como a cobertura de métodos é menos precisa: temos um 100% que poderia significar muito, mas se apenas 13% das linhas foram testadas, certamente existem cenários dos métodos que não foram testados. Logo, faz muito mais sentido apresentar 13% como medida de cobertura de testes dessa classe.

Qual deve ser a cobertura mínima?

Não existe consenso quanto a uma quantidade mínima. Uma cobertura mínima ideal pode variar de acordo com o tipo de projeto, de linguagem de programação utilizada e das bibliotecas/*frameworks* usados. Porém, os números mais citados estão **entre 70% e 90%** de cobertura de linhas. Por exemplo, o **SonarQube**, que é uma ferramenta de análise de qualidade de código, tem como valor padrão mínimo de cobertura de código **80%**. Falaremos brevemente sobre essa ferramenta no Apêndice B.

OPINIÃO PROFISSIONAL

"Sempre devemos criar testes de unidade! Imperdoável não haver testes. Muitas vezes testes de unidade são difíceis ou muito custosos, e podem ser usados testes funcionais ou de integração."

Eduardo Guerra, doutor e mestre em Engenharia Eletrônica e Computação pelo ITA.

"Teste de unidade é essencial! Os testes de unidade são a linha de frente de qualquer batalha por um código mais limpo!"

Camila Achutti, mestra e doutoranda em Ciência da Computação pela USP.

Neste capítulo, vimos a importância de criar testes automatizados de unidade e uma pequena demonstração de como fazê-los. No próximo capítulo, serão apresentadas técnicas de Código Limpo para a criação de classes.

Capítulo 9
Classes

Alguns amigos de João Kacau foram visitá-lo, pois era seu aniversário. Para comemorar seu aniversário com classe: alguns levaram até garrafas de vinho chileno; outros levaram carnes nobres para assar, e outros ainda levaram ingredientes e massa para fazerem pizzas.

Tudo era alegria, tudo era empolgação. Até que chegou a hora de abrir o vinho. Perguntaram a João onde estava o saca-rolhas. Ele disse que estava no armário branco da cozinha. Os que foram lá levaram quase 10 minutos para achar. Então chegou o momento de começar a preparar o churrasco. Perguntaram onde estavam o carvão, as facas, sal etc. João disse que estava no armário branco da cozinha. Os que foram lá levaram quase 15 minutos para achar tudo. Por fim, chegou a hora de fazer as pizzas. Perguntaram onde estavam a forma, o cortador, a lenha para o forno etc. João disse que estava no armário branco da cozinha. Os que foram lá levaram quase 15 minutos para achar tudo.

Talvez você esteja pensando que os amigos de João são muito lentos em achar as coisas. Porém, notou um pequeno detalhe? Tudo estava em um suposto armário branco da cozinha. E sabe o que os amigos disseram sobre esse armário?

1. Não havia só um armário branco, e sim dois;
2. Esses armários tinham, cada um, apenas um compartimento;
3. Não havia uma organização dentro dos armários. Os tipos de materiais não estavam agrupados. Tudo estava misturado;
4. Os armários estavam sujos: havia pacotes vazios e até teias de aranha.

Pois, tal como a cozinha de João Kacau, assim podem ser nossas **classes** quando programamos orientados a objetos: podem fazer

coisas demais, ser confusas, cheias de coisas inúteis. Neste capítulo, vamos ver as orientações de Código Limpo para **classes**.

9.1 Organização de uma classe por tipo de elemento

Uma classe pode possuir construtores, atributos de instância e classe, constantes e métodos. A ordem desses elementos não influencia no funcionamento de uma classe, porém uma classe com elementos bem ordenados pode facilitar muito a sua leitura e manutenção. Vejamos a seguir uma proposta de ordem dos elementos de uma classe, considerando os elementos mais usados no cotidiano de projetos orientados a objetos.

1. **Constantes estáticas públicas**;
2. **Constantes estáticas privadas**;
3. **Atributos de instância privados**;
4. **Construtor vazio público** — O construtor vazio também é chamado de **construtor padrão**;
5. **Demais construtores públicos** — Começando pelos que têm menos parâmetros;
6. **Métodos públicos**;
7. **Métodos privados** — Na obra original *Código Limpo*, a orientação é que cada método privado venha logo após o método público que o invoca.

Porém, existe uma outra forma de organização bem difundida, que é deixar todos os métodos privados após os públicos, começando pelos que têm menor dependência. Por exemplo: se um dos métodos privados não invoca nenhum dos demais privados para funcionar, ele deve ser o primeiro. O último método privado deve ser o que invoca mais métodos privados da classe.

Existem outros elementos que podem ser usados em classes orientadas a objetos, mas são bem menos comuns. Se formos considerar todos os elementos possíveis, a ordem seria:

1. **Constantes estáticas públicas**;
2. **Constantes estáticas protegidas**;
3. **Constantes estáticas privadas**;
4. **Atributos de classe públicos**;
5. **Atributos de classe protegidos**;
6. **Atributos de classe privados**;
7. **Atributos de instância públicos**;
8. **Atributos de instância protegidos**;
9. **Atributos de instância privados**;
10. **Métodos estáticos públicos**;
11. **Métodos estáticos protegidos**;
12. **Métodos estáticos privados**;
13. **Construtor vazio público** — O construtor vazio também é chamado de **construtor padrão**;
14. **Construtor vazio protegido** — O construtor vazio também é chamado de *construtor padrão*;
15. **Construtor vazio privado** — O construtor vazio também é chamado de *construtor padrão*;
16. **Demais construtores públicos** — Começando pelos que têm menos parâmetros;
17. **Demais construtores protegidos** — Começando pelos que têm menos parâmetros;
18. **Demais construtores privados** — Começando pelos que têm menos parâmetros;
19. **Métodos públicos**;
20. **Métodos protegidos**;
21. **Métodos privados** — Usando a abordagem de cada método privado ficar logo após o método público/protegido que o invoca OU todos os métodos privados após todos os públicos e protegidos, começando pelos que têm menor dependência.

Atributos de instância x atributos de classe

Há um conceito que costuma causar confusão em iniciantes na Programação Orientada a Objetos: a diferença entre atributos de **instância** e de **classe**.

Um **atributo de instância** é aquele cujo valor é **único** para cada instância de uma classe. Analogias com o mundo real seriam sua impressão digital ou seu CPF, por exemplo.

Um **atributo de classe** é aquele cujo valor é o mesmo **para todas** as instâncias de uma classe. Uma analogia com o mundo real seria o tempo mínimo para um homem se aposentar por idade em um determinado país: se esse valor muda por lei, passa a valer para todos os homens daquele país.

Em Java, um atributo de instância é todo aquele que não é estático (`static`). Logo, se um atributo é estático, é um atributo de classe. O fato de ser `public` , `private` ou `protected` não influencia nessa questão. Vejamos o código Java a seguir.

```
public class Cofrinho {

  public static Integer MAXIMO_DE_DEPOSITOS = 20;
  // Atributo de classe. Todo "Cofrinho" do projeto terá o mesmo valor de "MAXIMO_DE_DEPOSITOS", ou seja, 20.

  private Double valorDepositado;
  // Atributo de instância. O valor de "valorDepositado" de uma instância de "Cofrinho" não influencia nas demais instâncias.

}
```

Em Python, todo atributo declarado fora de métodos, ou seja, diretamente no corpo da classe, é um atributo de classe. Quando um atributo é citado a partir do `self` , é um atributo de instância. Vejamos o código Python a seguir.

```
class Cofrinho():

   maximo_de_depositos = 20
```

```
    # Atributo de classe. Todo "Cofrinho" do projeto terá o mesmo valor de
"maximo_de_depositos", ou seja, 20.

    def __init__(self, valor_inicial):
        self._valor_depositado = valor_inicial
        # Atributo de instância. O valor de "valor_depositado" de uma
instância de "Cofrinho" não influencia nas demais instâncias.
```

Conforme explicado com mais detalhes no *Apêndice A — Convenções de código*, o nome de um atributo de classe segue convenções diferentes para Java e Python:

- **Java**: MACRO_CASE;
- **Python**: snake_case.

Essa ordem de elementos proposta tem como princípio começar pelos elementos de maior escopo em um projeto, por isso começamos com o que é estático. Então, vamos dos elementos públicos para os privados. Isso se deve ao fato de que uma classe normalmente existe para ser usada por outras classes. Por isso, elementos visíveis às demais classes merecem uma atenção primária no código de uma classe. Já a ordem de atributos primeiro e métodos depois é muito mais para ficar parecida com a ordem que vemos em diagramas de classe da **UML** (*Unified Modeling Language*, uma linguagem de notação que permite modelar sistemas, principalmente baseados em software).

9.2 Encapsulamento

O encapsulamento é um dos conceitos mais importantes da Programação Orientada a Objetos. Para implementá-lo, os atributos que queremos manter seguros/protegidos devem ser **privados** (ou, em alguns casos, **protegidos**) e recuperados e/ou alterados apenas por meio de algum método ou construtor públicos.

Sempre que possível, devemos usar esse conceito. Porém, isso não deve ser uma obsessão. Há situações em que atributos e métodos podem ser protegidos (*protected*) ou com nível de acesso padrão, como para facilitar a criação de testes unitários, por exemplo.

9.3 Olha que classe grande! Cuidado com a baixa coesão!

Se você já leu o capítulo sobre funções, viu que um código limpo exige que funções tenham alta coesão. O mesmo vale para classes: quanto menos coisas diferentes uma classe faz, maior a coesão e mais limpo é seu código.

Nível de coesão x nível de acoplamento

Assim como explicado sobre funções, uma classe com baixa **coesão** é aquela que tem muitas responsabilidades, ou seja, que se propõe a resolver muitos problemas. Logo, quanto mais especialista/específica for uma função, maior sua coesão.

Temos que sempre buscar um alto nível de coesão nas classes. É uma consequência quase certa em classes de baixa coesão o nível de acoplamento entre as classes ser maior.

Acoplamento é o nível de dependência entre as classes. Quando menor o nível de acoplamento, melhor é um código, pois reduz as chances de efeitos colaterais em mudanças de código. Claro que um projeto não pode ter 0% de acoplamento, mas devemos buscar o menor nível possível.

A baixa coesão não é o único motivo que leva ao alto acoplamento, mas é um dos principais. Outro grande vilão é o baixo nível de abstração (veja no tópico de SOLID, a seguir). Em um código limpo, as classes possuem **alta coesão e baixo acoplamento**. É possível ver a relação entre coesão e acoplamento na figura a seguir.

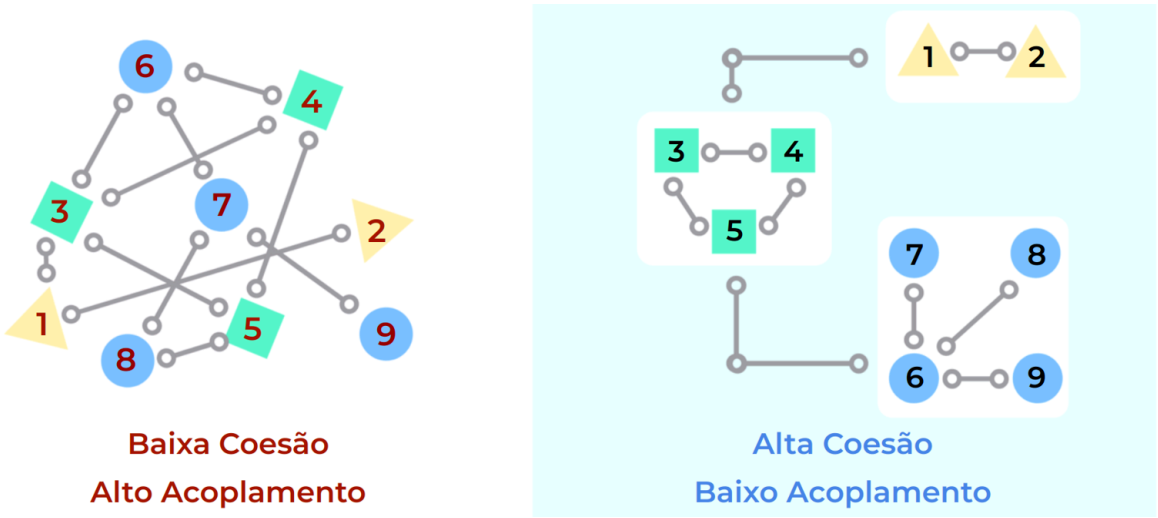


Figura 9.1: Níveis de coesão x acoplamento em classes.

Segundo os dois cenários presentes na figura, quantas classes poderiam ser impactadas ao alterarmos a classe 3? Em um cenário de baixa coesão e alto acoplamento, haveria chances de impactarmos as classes 1, 4, 5 e 6. Já no cenário de alta coesão e baixo acoplamento, as classes que talvez fossem impactadas seriam apenas a 4 e a 5.

Se consideramos que o polígono da classe (triângulo, quadrado e círculo) representa a categoria ou a camada de cada classe, é possível então observar que, no cenário de baixa coesão e alto acoplamento, não há limites bem definidos entre as categorias/camadas. Já no cenário de alta coesão e baixo acoplamento existe uma organização bem melhor em categorias/camadas.

9.4 SOLID (resumo)

Além dos pontos já expostos neste capítulo, para que uma classe tenha um código de qualidade, ele deve ter sido implementado

seguindo um conjunto de princípios conhecidos como **SOLID**. A seguir, um resumo sobre ele.

Como e quando surgiram esses princípios

Os princípios SOLID foram propostos por Robert C. Martin em 2000, que escreveu sobre eles em uma série de artigos chamada *Design Principles and Design Patterns*. Uma curiosidade é que o acrônimo SOLID não foi cunhado por Robert Martin, e sim por Michael Feathers, anos depois, ao perceber que as primeiras letras dos cinco princípios formavam essa palavra (que, em português, quer dizer *sólido*).

Quais são os princípios SOLID

Os cinco princípios SOLID foram propostos para o paradigma da Programação Orientada a Objetos, São eles:

- **Princípio da Responsabilidade Única**;
- **Princípio aberto-fechado**;
- **Princípio da Substituição de Liskov**;
- **Princípio da Segregação de Interface**;
- **Princípio da Inversão de Dependência**;

A seguir, um resumo sobre cada um deles.

9.5 Princípio da Responsabilidade Única

Nome original: ***Single Responsibility Principle*** **(SRP)**.

Imagine um sistema relativamente simples, como uma calculadora básica que armazena o histórico das operações realizadas. Seria possível implementá-lo criando apenas uma classe? Sim, porém, *tudo* estaria nela: códigos para validação das entradas; códigos para realização das operações matemáticas; códigos para

armazenamento no histórico; códigos de recuperação de operação do histórico e códigos necessários para a interface com o usuário. Sendo assim, reflita:

- Quão grande ficaria a classe?
- Quantos tipos de códigos diferentes teríamos em um mesmo arquivo?
- Como seria encontrar e tratar erros nessa classe?

As respostas na reflexão não foram muito animadoras, certo? Isso porque essa classe teria **muitas responsabilidades**. E é esse tipo de situação que o SRP (*Single Responsibility Principle*, ou Princípio da Responsabilidade Única) orienta que **não aconteça**. Segundo esse princípio, uma classe ou método que possui muitas responsabilidades muito provavelmente terá os seguintes problemas:

- Será, como se diz popularmente, como um *pato*: faz várias coisas, mas nenhuma muito bem;
- Terá difícil manutenção: achar e corrigir um bug ou implementar uma nova funcionalidade serão tarefas cada vez mais difíceis com o tempo;
- Será mais difícil criar testes automatizados para classes e métodos assim.

Para ilustrar a situação de uso ou não uso do SRP, basta lembrarmos de situações reais fora da tecnologia. Por exemplo, o conhecido canivete suíço. À primeira vista, parece prático: *nossa, várias ferramentas em um mesmo objeto!* Só que... Nem sempre. Não podemos usar a faca enquanto outra pessoa usa o saca-rolhas. Cada ferramenta é um tanto desconfortável de usar. É praticamente impossível substituir apenas uma das ferramentas, caso só uma quebre. Por fim, verificamos que, para um profissional que lida com as ferramentas existentes nesse canivete, a melhor opção seria mesmo uma caixa de ferramentas, onde elas ficam organizadas e cada uma tem a sua responsabilidade.

Em resumo, esse princípio orienta que *classes e métodos tenham apenas uma responsabilidade*, para que tenham manutenção mais fácil e sejam mais fáceis de testar. Uma classe/método deve ter apenas uma razão, um motivo para ser alterada.

9.6 Princípio aberto-fechado

Nome original: ***Open-closed Principle*** **(OCP)**.

A um sistema de leitura que usa inteligência artificial para ler, é solicitado que seja lida uma placa. Ele responde que não entende o que está escrito nela. Qual seria o problema? Seria o dispositivo de captura (câmera) ou a placa (suja, riscada, longe)? E se ambos estiverem sem problemas? E se a placa estava bem próxima da câmera e em um ambiente bem iluminado? Lembrei de um detalhe: a placa está escrita em russo, e o sistema foi ensinado a ler em português somente.

Para resolver o problema descrito, precisamos trocar a câmera? Precisamos trocar ou melhorar a biblioteca de OCR (tradução imagem para texto)? Na verdade, não há nada de errado com a câmera ou o OCR. O sistema precisa de uma **nova habilidade**, não concorda? A habilidade de ler em russo.

Essa situação serve para ilustrar o **OCP (*Open-closed Principle*, ou Princípio aberto-fechado)**, o qual orienta que devemos programar de forma que nossas entidades (classes, módulos, métodos etc.) estejam **abertas** para mudanças por meio de extensões e **fechadas** para alterações diretas em seus próprios códigos-fonte.

Não é necessário trocar ou melhorar o OCR. O sistema consegue, sem alterações invasivas nele, aprender um novo idioma. Tampouco

é necessário mexer em algo na câmera, que é onde começa a leitura. Vejamos um conjunto de interfaces, classes e métodos que implementariam esse princípio para a situação do exemplo do leitor da placa.

Interface `Idioma` em Java:

```java
public interface Idioma {

    String ler(String texto);

}
```

Interface `Idioma` em Python:

```python
# Na verdade, em Python não existem interfaces
from abc import abstractmethod

class Idioma :

    @abstractmethod
    def ler(self, texto) -> str:
        pass
```

Classe `IdiomaPortugues` em Java:

```java
public class IdiomaPortugues implements Idioma {

    public String ler(String texto) {
        // implementação da leitura em português
    }

}
```

Classe `IdiomaPortugues` em Python:

```python
class IdiomaPortugues(Idioma) :
    def ler(self, texto: str) -> str:
        # implementação da leitura em português
```

Classe `Cerebro` em Java:

```java
public class Cerebro {

    public String ler(String texto, Idioma idiomaTexto) {
        return idiomaTexto.ler(texto);
    }

}
```

Classe `Cerebro` em Python:

```python
class Cerebro:

    def ler(self, texto: str, idioma_texto: Idioma) -> str:
        return idioma_texto.ler(texto)
```

Então, passa a ser necessário que os leitores no programa consigam ler em russo. Basta criarmos a classe **IdiomaRusso**. Veja a seguir.

Java:

```java
public class IdiomaRusso implements Idioma {

    public String ler(String texto) {
        // implementação da leitura em russo
    }

}
```

Python:

```python
class IdiomaRusso(Idioma) :
    def ler(self, texto: str) -> str:
        # implementação da leitura em russo
```

Para ler textos em vários idiomas, o código ficaria como o exemplo a seguir:

Java:

```
Cerebro bilingue = new Cerebro();

Idioma idioma1 = new IdiomaPortugues();
Idioma idioma2 = new IdiomaRusso();

System.out.println(bilingue.ler("Bom dia, flor do dia!", idioma1));
System.out.println(bilingue.ler("Доброе утро, цветок дня!", idioma2));
```

Python:

```
bilingue = Cerebro()

idioma1 = IdiomaPortugues()
idioma2 = IdiomaRusso()

print(bilingue.ler("Bom dia, flor do dia!", idioma1))
print(bilingue.ler("Доброе утро, цветок дня!", idioma2))
```

Se `Cerebro` fosse uma classe, poderia ter um método como `ler(String texto, Idioma idiomaTexto)`, no qual o tipo do segundo parâmetro (`Idioma`) poderia ser uma *interface* ou uma *classe abstrata*. A `Idioma` teria um método abstrato `ler(String texto)`, invocado dentro da `ler()` de `Cerebro`. Com essa arquitetura, poderíamos criar quantas implementações de idiomas quiséssemos. Em nosso exemplo, só existia a classe `IdiomaPortugues`, que implementava um idioma. Para fazer a pessoa ler em russo, bastaria criarmos a implementação `IdiomaRusso`. Se não tivéssemos seguido o **OCP**, nossa classe `Cerebro` teria métodos como `lerPortugues()` / `ler_portugues()` e `lerRusso()` / `ler_russo()`, o que implicaria criar novos métodos sempre que fosse necessário ler em um novo idioma.

Em resumo, esse princípio orienta que suas entidades estejam programadas de tal forma que seja possível acrescentar novas funcionalidades sem que se crie código-fonte em entidades já existentes. Em vez disso, deve ser possível implementá-las criando novas entidades.

9.7 Princípio da Substituição de Liskov

Nome original: ***Liskov Substitution Principle*** **(LSP)**.

Vamos continuar com o exemplo usado anteriormente, o da leitura em vários idiomas em um mesmo cérebro. Imagine que fizéssemos pequenas alterações nas classes `IdiomaPortugues` e `IdiomaRusso`.

Classe `IdiomaPortugues` em Java:

```java
public class IdiomaPortugues implements Idioma {

    public String ler(String texto) {
        // implementação da leitura em português
    }

    public String lerPt(String texto) {
        // leitura em português de Portugal
    }

    public String lerBr(String texto) {
        // leitura em português do Brasil
    }
}
```

Classe `IdiomaPortugues` em Python:

```python
class IdiomaPortugues(Idioma) :
    def ler(self, texto: str) -> str:
        # implementação da leitura em português

    def ler_pt(self, texto: str) -> str:
        # leitura em português de Portugal

    def ler_br(self, texto: str) -> str:
        # leitura em português do Brasil
```

Classe `IdiomaRusso` em Java:

```
public class IdiomaRusso implements Idioma {

    public String ler(String texto) {
        // implementação da leitura em russo
    }

    public String lerApos4Vodkas(String texto) {
        // leitura em russo simulando já ter tomado quatro garrafas de
vodka
    }

}
```

Classe `IdiomaRusso` em Python:

```
class IdiomaRusso(Idioma) :
    def ler(self, texto: str) -> str:
        # implementação da leitura em russo

    def ler_apos_4_vodkas(self, texto: str) -> str:
        # leitura em russo simulando já ter tomado quatro garrafas de
vodka
```

Note que acrescentamos dois novos métodos em `IdiomaPortugues` e um novo método em `IdiomaRusso`. Segundo o LSP (*Liskov Substitution Principle*, ou, em português, Princípio da Substituição de Liskov*), **isso** não é recomendável**, pois pode trazer surpresas bem desagradáveis. Vejamos um exemplo disso no próximo código.

Java:

```
Cerebro bilingue = new Cerebro();

Idioma idioma1 = new IdiomaPortugues();
Idioma idioma2 = new IdiomaRusso();

System.out.println(bilingue.ler("Bom dia, flor do dia!", idioma1));
System.out.println(bilingue.ler("Доброе утро, цветок дня!", idioma2));

System.out.println(idioma1.lerBr("Vá para o final da fila"));
```

```
System.out.println(idioma1.lerPt("Vá para o rabo da bicha"));
System.out.println(idioma2.lerApos4Vodkas("O rato roeu a roupa do rei de
Roma"));
```

Python:

```
bilingue = Cerebro()

idioma1 = IdiomaPortugues()
idioma2 = IdiomaRusso()

print(bilingue.ler("Bom dia, flor do dia!", idioma1))
print(bilingue.ler("Доброе утро, цветок дня!", idioma2))

print(idioma1.ler_br("Vá para o final da fila"))
print(idioma1.ler_pt("Vá para o rabo da bicha"))
print(idioma2.ler_apos_4_vodkas("O rato roeu a roupa do rei de Roma"))
```

Percebeu o problema desse código? Ele compilaria? Daria erro em tempo de execução? Na verdade, ele *sequer iria compilar*, pois o objeto `idioma2` está declarado como `Idioma`, que só possui o método `ler(String texto)`. Assim, a última linha ficaria marcada como erro pelo compilador.

Talvez você pense: "*Mas isso é fácil de resolver! Basta declarar a `idioma2` como `IdiomaRusso`!*". De fato, resolveria... Porém, vamos supor que o programa evoluísse de tal forma que o tipo de idioma é determinado em tempo de execução, e não mais direto no código. Vejamos isso no próximo código:

Java:

```
Cerebro bilingue = new Cerebro();

Idioma idiomaX = // algum código que retorna o tipo em tempo de execução

System.out.println(bilingue.ler("Bom dia, flor do dia!", idiomaX));

System.out.println(idiomaX.lerBr("Vá para o final da fila"));
```

Python:

```
bilingue = Cerebro()

idiomaX = # algum código que retorna o tipo em tempo de execução

print(bilingue.ler("Bom dia, flor do dia!", idiomaX))

print(idiomaX.ler_br("Vá para o final da fila"))
```

Note que esse código precisa que o objeto `idiomaX` seja declarado como `Idioma`, pois somente em tempo de execução será instanciado como algumas de suas implementações. Com isso, a última linha provocaria um erro em tempo de execução, uma vez que os métodos `lerBr()` / `ler_br()` e `lerPt()` / `ler_pt()` só existem na implementação `IdiomaPortugues`.

Para evitar esses tipos de problemas, o LSP orienta que o *contrato* (métodos públicos) de uma classe devem ser o mesmo de sua *superclasse* (ou *classe base*) ou *interface* implementadas. Se essa orientação for seguida, podemos substituir objetos de uma *superclasse (ou interface) comum*, mesmo em tempo de execução, sem prejuízos ao programa.

Não apenas mudanças explícitas de contrato, como novos métodos públicos, ferem esse princípio. Caso as *pré-condições* do contrato da subclasse forem mais fortes que as da superclasse e/ou as *pós-condições* forem mais fracas, esse princípio também é violado.

As **pré-condições** são os critérios para que um método público funcione. Vamos analisar o seguinte cenário:

1. O método `ler()` de `IdiomaPortugues` não possui nenhuma validação;
2. Criamos a classe `IdiomaPortuguesBrasileiro`, que é subclasse de `IdiomaPortugues`;
3. No método `ler()` de `IdiomaPortuguesBrasileiro`, incluímos a validação de pelo menos 10 caracteres para seu argumento.

Caso violada, é lançada uma exceção.

Nesse cenário, violamos o LSP porque as pré-condições da subclasse são mais fortes do que as da superclasse. Isso porque o comportamento de um objeto do tipo `Idioma` pode ser muito diferente ao ter seu método `ler()` invocado, podendo lançar uma exceção somente caso um objeto for do tipo `IdiomaPortuguesBrasileiro` em tempo de execução.

As **pós-condições** (ou **condições posteriores**) são os possíveis retornos de métodos públicos. Vamos analisar o seguinte cenário:

1. O método `ler()` de `IdiomaPortugues` pode retornar um texto (`String` em Java ou `str` em Python) vazio ou nulo (`null` em Java ou `None` em Python);
2. Criamos a classe `IdiomaPortuguesBrasileiro`, que é subclasse de `IdiomaPortugues`;
3. No método `ler()` de `IdiomaPortuguesBrasileiro`, só existe a possibilidade de o método retornar um texto com pelo menos um caractere, mas nunca vazio ou nulo.

Nesse cenário, violamos o LSP porque as pós-condições da subclasse são mais fracas que as da superclasse. Isso porque o método `ler()` tem menos tipos de resultados diferentes na subclasse do que na superclasse.

Outra coisa que pode violar o LSP nas **pós-condições** é a forma como lançamos exceções: uma subclasse não pode lançar exceções que sua superclasse não lance. Vamos analisar o seguinte cenário:

1. O método `ler()` de `IdiomaPortugues` pode lançar uma única exceção: `IllegalArgumentException` em Java ou `ValueError` em Python;
2. Criamos a classe `IdiomaPortuguesBrasileiro`, que é subclasse de `IdiomaPortugues`;
3. No método `ler()` de `IdiomaPortuguesBrasileiro`, incluímos regras que possibilitam uma segunda exceção:

`ArrayIndexOutOfBoundsException` em Java ou `IndexError` em Python.

Nesse cenário, violamos o LSP porque todo código que usa objetos do tipo `IdiomaPortugues` está preparado para lidar apenas com exceções de um tipo. Porém, no caso de a classe ser `IdiomaPortuguesBrasileiro`, em tempo de execução, uma segunda e não esperada exceção pode ser lançada, o que pode levar a um *crash* na aplicação.

Em resumo: esse princípio orienta que classes não tenham métodos públicos que não existem na sua superclasse (ou interface que implementa), para evitar erros de compilação e em tempo de execução.

> **Uma curiosidade:** ao ler o nome *Liskov*, que tipo de pessoa você imaginou? Quando dou aulas de SOLID e faço essa pergunta a meus alunos e alunas, quase sempre pensam em um cara russo de idade um tanto avançada. Na verdade, refere-se à norte-americana **Barbara Liskov**, a primeira mulher a ter um PhD em Ciência da Computação nos EUA e detentora de vários prêmios relevantes devido às suas contribuições para a Computação. Esse princípio é de sua autoria, por isso Robert C. Martin usou o nome dela para nomear o LSP.

9.8 Princípio da Segregação de Interface

Nome original: ***Interface Segregation Principle*** **(ISP)**.

Continuemos na evolução do programa de `Cerebro` usado nos últimos princípios. Vamos supor que mais e mais funcionalidades serão acrescentadas ao `Cerebro`, como escrever, desenhar, compor

músicas etc. Bastaria criar métodos nessa classe, certo? Seria essa uma boa solução?

Imagine que surjam necessidades de tipos diferentes de cérebros e habilidades, como nestes exemplos: o *cérebro humano* possui a habilidade de *cantar*, assim como o *cérebro de bem-te-vi*; o *cérebro humano* tem a habilidade de *compor músicas*, diferente do *cérebro de elefante*, que não a tem. Percebe que temos diferentes tipos de cérebros que compartilham ou não algumas habilidades?

Uma forma de usar a orientação a objetos para implementar essa necessidade seria tornar a `Cerebro` abstrata, como a seguir:

Java:

```java
public abstract class Cerebro {

    public abstract String ler(String texto, Idioma idiomaTexto);

    public abstract String cantar();

    public abstract String comporMusica();

    public abstract String voar();
}
```

Python:

```python
from abc import ABC, abstractmethod

class Cerebro(ABC):

    @abstractmethod
    def ler(self, texto: str, idioma_texto: Idioma) -> str:
        pass

    @abstractmethod
    def cantar(self) -> str:
        pass
```

```
    @abstractmethod
    def compor_musica(self) -> str:
        pass

    @abstractmethod
    def voar(self) -> str:
        pass
```

Dessa forma, as subclasses de `Cerebro` implementariam os métodos necessários e deixariam os outros vazios ou com lançamento de exceções do tipo `UnsupportedOperationException`, por exemplo. Vejamos como ficaram as classes `CerebroHumano` e `CerebroBemtevi`, a seguir.

Classe `CerebroHumano` em Java:

```
public class CerebroHumano extends Cerebro {

    public String ler(String texto, Idioma idiomaTexto) {
        // implementação
    }

    public String cantar(){
        // implementação
    }

    public String comporMusica(){
        // implementação
    }

    public String voar(){
        // vazio ou lança uma exceção
    }
}
```

Classe `CerebroHumano` em Python:

```
class CerebroHumano(Cerebro):

    def ler(self, texto: str, idioma_texto: Idioma) -> str:
```

```
        # implementação

    def cantar(self) -> str:
        # implementação

    def compor_musica(self) -> str:
        # implementação

    def voar(self) -> str:
        # vazio ou lança uma exceção
```

Classe `CerebroBemtevi` em Java:

```
public class CerebroBemtevi extends Cerebro {

    public String ler(String texto, Idioma idiomaTexto) {
        // vazio ou lança uma exceção
    }

    public String cantar(){
        // implementação
    }

    public String comporMusica(){
        // vazio ou lança uma exceção
    }

    public String voar(){
        // implementação
    }
}
```

Classe `CerebroBemtevi` em Python:

```
class CerebroBemtevi(Cerebro):

    def ler(self, texto: str, idioma_texto: Idioma) -> str:
        # vazio ou lança uma exceção

    def cantar(self) -> str:
```

```
        # implementação

    def compor_musica(self) -> str:
        # vazio ou lança uma exceção

    def voar(self) -> str:
        # implementação
```

Notou que a `CerebroHumano` ficou com um método sem utilidade e a `CerebroBemtevi`, com dois? Ou seja, métodos foram implementados simplesmente para ficarem inúteis em algumas situações. É esse tipo de situação que o **ISP (*Interface Segregation Principle*, ou Princípio da Segregação de Interface)** orienta a evitar, uma vez que ele diz que *classes não devem ser forçadas a depender de métodos que não usam*.

Para seguir a orientação do ISP, poderíamos refatorar nosso código deixando-o da seguinte forma:

Interface `CerebroLeitor` em Java:

```
public interface CerebroLeitor {

    String ler(String texto, Idioma idiomaTexto);

}
```

Interface `CerebroLeitor` em Python:

```
from abc import ABC, abstractmethod

class CerebroLeitor(ABC):

    @abstractmethod
    def ler(self, texto: str, idioma_texto: Idioma) -> str:
        pass
```

Interface `CerebroCantor` em Java:

```java
public interface CerebroCantor {

    String cantar();

}
```

Interface `CerebroCantor` em Python:

```python
from abc import ABC, abstractmethod

class CerebroCantor(ABC):

    @abstractmethod
    def cantar(self) -> str:
        pass
```

Interface `CerebroCompositor` em Java:

```java
public interface CerebroCompositor {

    String comporMusica();

}
```

Interface `CerebroCompositor` em Python:

```python
from abc import ABC, abstractmethod

class CerebroCompositor(ABC):

    @abstractmethod
    def compor_musica(self) -> str:
        pass
```

Interface `CerebroVoador` em Java:

```java
public interface CerebroVoador {

    String voar();
}
```

Interface `CerebroVoador` em Python:

```
from abc import ABC, abstractmethod

class CerebroVoador(ABC):

    @abstractmethod
    def voar(self) -> str:
        pass
```

Observe que eliminamos a classe abstrata `Cerebro` e a dividimos em **interfaces**, cada uma muito específica. Continuemos a refatoração.

Classe `CerebroHumano` em Java:

```
public class CerebroHumano implements CerebroLeitor, CerebroCantor,
CerebroCompositor {

    public String ler(String texto, Idioma idiomaTexto) {
        // implementação
    }

    public String cantar(){
        // implementação
    }

    public String comporMusica(){
        // implementação
    }

}
```

Classe `CerebroHumano` em Python:

```
class CerebroHumano(CerebroLeitor, CerebroCantor, CerebroCompositor):

    def ler(self, texto: str, idioma_texto: Idioma) -> str:
        # implementação

    def cantar(self) -> str:
```

```
        # implementação

    def compor_musica(self) -> str:
        # implementação
```

Classe `CerebroBemtevi` em Java:

```
public class CerebroBemtevi extends CerebroCantor, CerebroVoador {

    public String cantar(){
        // implementação
    }

    public String voar(){
        // implementação
    }
}
```

Classe `CerebroBemtevi` em Python:

```
class CerebroBemtevi(CerebroCantor, CerebroVoador):

    def cantar(self) -> str:
        # implementação

    def voar(self) -> str:
        # implementação
```

As classes `CerebroHumano` e `CerebroBemtevi` agora implementam uma ou mais interfaces, de acordo com os métodos que realmente precisam. Isso porque criamos interfaces **segregadas**, ou seja, bem específicas.

Em resumo: as interfaces devem ser as mais específicas possíveis (ou seja, ter o mínimo de métodos) para que as classes usem as interfaces necessárias e para que não sejam forçadas a implementar métodos dos quais não precisam.

9.9 Princípio da Inversão de Dependência

Nome original: ***Dependency Inversion Principle*** **(DIP)**.

Ligar um carro, se você tem a chave, é muito fácil, não? Mas, já parou para pensar no que ocorre dentro do carro quando você faz o singelo ato de virar a chave? Pense em quantos sistemas eletrônicos entram em ação, e depois sistemas mecânicos... Em carros contemporâneos, não tenha dúvida de que estamos falando de dezenas de ações que o veículo faz por nós nesse momento.

E você sabe como os primeiros carros comercializados eram ligados? Era mais ou menos assim: o motorista pegava uma *manivela*, ia para a frente do carro, enfiava a manivela lá e a girava bem rápido até o motor funcionar. Alguns detalhes: a manivela era bem pesada e girá-la também demandava muita força. Quais as desvantagens da técnica anterior? Era mais difícil, era mais demorado, era mais perigoso.

Poderíamos pensar em vários outros exemplos de modernização que nos trariam paralelos como este: fogão a lenha x fogão moderno; pedir comida por telefone x pedir comida por app de celular; pagar usando cheque x pagar usando cartão de débito etc.

Sabe o que foi feito em todos esses exemplos? Uma coisa chamada **abstração**. Foram abstraídos detalhes, ações que seriam repetitivas, demoradas ou até perigosas de tal forma que as pessoas fizessem o mínimo necessário para realizar uma determinada tarefa de forma mais fácil, rápida e segura.

É em defesa do uso de **abstração** em programação que surgiu o **DIP (*Dependency Inversion Principle*, ou Princípio da Inversão de Dependência)**. Assim como o usuário final de um programa tem benefícios quando usufrui de abstração nas funcionalidades, programadores têm benefícios quando seu código-fonte possui abstrações que facilitam a evolução e manutenção de sistemas.

Retomando nosso exemplo de sistema de cérebros, poderíamos considerar que, **para qualquer ação que um cérebro faça**, deve-se verificar que ele está *acordado* e *pronto*. Se não estiver, primeiro deve *acordar* e depois *ficar pronto*. Se já estiver *acordado*, deve apenas *ficar pronto*. Uma forma ferir o **DIP** nesse exemplo seria um código como este:

Java:

```
public class CerebroBemtevi extends CerebroCantor, CerebroVoador {

    private boolean acordado = false;
    private boolean pronto = false;

    public void acordar() {
        acordado = true;
    }

    public void ficarPronto() {
        pronto = true;
    }

    public boolean isAcordado() {
        return acordado;
    }

    public boolean isPronto() {
        return pronto;
    }

    // métodos cantar() e voar()

}
```

Python:

```
class CerebroBemtevi(CerebroCantor, CerebroVoador):

    def __init__(self):
        self.__acordado = False
```

```
        self.__pronto = False

    def acordar(self):
        self.__acordado = True

    def ficar_pronto(self):
        self.__pronto = True

    def is_acordado(self) -> bool:
        return self.__acordado

    def is_pronto(self) -> bool:
        return self.__pronto

    # métodos cantar() e voar()
```

A seguir, um código qualquer que use `CerebroBemtevi`:

Java:

```
CerebroBemtevi cerebro = new CerebroBemtevi();

if (!cerebro.isAcordado()) {
    cerebro.acordar();
}
if (!cerebro.isPronto()) {
    cerebro.ficarPronto();
}
System.out.println(cerebro.cantar());

// nada garante que, após cantar, o cérebro continue acordado e pronto,
portanto...
if (!cerebro.isAcordado()) {
    cerebro.acordar();
}
if (!cerebro.isPronto()) {
    cerebro.ficarPronto();
}
System.out.println(cerebro.voar());
```

Python:

```
cerebro = CerebroBemtevi()

if (not cerebro.is_acordado()):
    cerebro.acordar()

if (not cerebro.is_pronto()):
    cerebro.ficarPronto()

print(cerebro.cantar())

# nada garante que, após cantar, o cérebro continue acordado e pronto,
portanto...
if (not cerebro.is_acordado()):
    cerebro.acordar()

if (not cerebro.is_pronto()):
    cerebro.ficar_pronto()

print(cerebro.voar())
```

Note que, ao usar uma instância de `CerebroBemtevi`, invocamos várias funcionalidades dela para implementar a regra de que o cerebro precisar estar *acordado* e *pronto* para realizar qualquer ação. Você acha que ficou simples? Ficou rápido de fazer (imagine se fossem mais ações, além de *cantar* e *voar*)? Ficou seguro (quem garante que não vamos esquecer de fazer algum dos testes antes de uma ação)?

Para melhorar a implementação, podemos começar a dar o primeiro passo na implantação do **DIP** refatorando a `CerebroBemtevi`, como o código que segue.

Java:

```
public class CerebroBemtevi extends CerebroCantor, CerebroVoador {

    // mesmos atributos de instância de antes
```

```
    private void acordar() {
        acordado = true;
    }

    private void ficarPronto() {
        pronto = true;
    }

    public String cantar(){
        if (!acordado) {
            acordar();
        }
        if (!pronto) {
            ficarPronto();
        }
        // implementação do cantar()
    }

    public String voar(){
        if (!acordado) {
            acordar();
        }
        if (!pronto) {
            ficarPronto();
        }
        // implementação do voar()
    }

}
```

Python:

```
class CerebroBemtevi(CerebroCantor, CerebroVoador):

    # mesmos construtor e atributos de instância de antes

    def __acordar(self):
        self.__acordado = True
```

```
    def __ficar_pronto(self):
        self.__pronto = True

    def cantar(self) -> str:
        if (not self.__acordado):
            self.__acordar()
        if (not self.__pronto):
            self.__ficar_pronto()
        # implementação do cantar()

    def voar(self) -> str:
        if (not self.__acordado):
            self.__acordar()
        if (not self.__pronto):
            self.__ficar_pronto()
        # implementação do voar()
```

A seguir, uma classe qualquer que use `CerebroBemtevi`, refatorada:

Java:

```
// em uma classe qualquer que usa CerebroBemtevi, refatorada

CerebroBemtevi cerebro = new CerebroBemtevi();

System.out.println(cerebro.cantar());
System.out.println(cerebro.voar());
```

Python:

```
cerebro = CerebroBemtevi()

print(cerebro.cantar())
print(cerebro.voar())
```

Veja que, como **abstraímos** as verificações de *está acordado* e *está pronto*, ficou muito mais simples, rápido e seguro pedir para um cérebro de bem-te-vi *cantar* e *voar*. Note o detalhe de que as ações

de *acordar* e *ficar pronto* agora são privadas na `CerebroBemtevi`, pois os clientes dessa classe não precisam saber desse detalhe que lhe foi *abstraído*.

Assim tivemos nosso primeiro passo rumo ao DIP; porém, ainda temos um certo volume de código repetido que não ficou legal, que são os `ifs` dentro dos `cantar()` e `voar()`. Poderíamos ficar ainda mais aderentes ao DIP, como pode ser visto no próximo código.

Classe `CerebroBemtevi` em Java:

```
public class CerebroBemtevi extends CerebroCantor, CerebroVoador {

    // mesmos atributos de instância de antes

    // métodos acordar() e ficarPronto() de antes

    private void prepararAcao() {
        if (!acordado) {
            acordar();
        }
        if (!pronto) {
            ficarPronto();
        }
    }

    public String cantar(){
        prepararAcao();
        // implementação do cantar()
    }

    public String voar(){
        prepararAcao();
        // implementação do voar()
    }

}
```

Classe `CerebroBemtevi` em Python:

```
class CerebroBemtevi(CerebroCantor, CerebroVoador):

    # mesmos construtor e atributos de instância de antes

    # métodos acordar() e ficar_pronto() de antes

    def __preparar_acao(self):
        if (not self.__acordado):
            self.__acordar()
        if (not self.__pronto):
            self.__ficar_pronto()


    def cantar(self) -> str:
        self.__preparar_acao()
        # implementação do cantar()

    def voar(self) -> str:
        self.__preparar_acao()
        # implementação do voar()
```

Nessa última versão de `CerebroBemtevi`, aumentamos a abstração da regra de *estar acordado e pronto antes de qualquer ação* na própria classe, jogando essa regra no método `prepararAcao()` / `preparar_acao()`, invocado no início dos métodos `cantar()` e `voar()` — o que pode ser feito no início de qualquer outro método que possamos implementar no futuro.

Vejamos outros exemplos de funcionalidades que deveriam estar abstraídas ao usarmos o **DIP** quando programamos, pois devem ser de responsabilidade de componentes específicos e que não devem ter seus detalhes espalhados e/ou repetidos pelo código-fonte:

- Controle de conexão e transação com bancos de dados;
- Comunicação com outros sistemas;
- Internacionalização;
- Conexão e comunicação com servidor de e-mail.

Na prática, muitas abstrações são feitas **com a adoção de bibliotecas** e **a criação de um módulo especialista com uma interface bem definida** (como que uma minibiblioteca interna).

Um exemplo bem concreto de DIP que é implementado por quase todas as linguagens de programação contemporâneas é o ***garbage collector***. Em linguagens mais antigas como a **C**, existia a preocupação com a retirada de variáveis da memória — caso contrário, elas ficariam lá indefinidamente e isso poderia causar estouro de memória. Atualmente, as linguagens têm um dispositivo chamado *garbage collector*, que faz com que variáveis não mais necessárias sejam retiradas automaticamente da memória. Ou seja, o gerenciamento de variáveis junto à memória foi **abstraído**, seguindo, assim, a orientação do DIP.

Em resumo: para tornar a programação mais simples, eficiente e até mais segura, devemos abstrair funcionalidades complexas, repetitivas ou perigosas.

OPINIÃO PROFISSIONAL

"A criação de classes pequenas, coesas e com um bom uso de encapsulamento é, para mim, o aspecto mais importante de um código em conjunto com funções e métodos coesos! Viabilizar coesão e priorizar manutenção em qualquer nível do seu código é primordial em ambientes profissionais de programação, característico por muito dinamismo."

Camila Achutti, mestra e doutoranda em Ciência da Computação pela USP.

"É imperdoável quando uma classe assume a responsabilidade de outra e quando faz várias coisas diferentes em um lugar só."

Eduardo Guerra, doutor e mestre em Engenharia Eletrônica e Computação pelo ITA.

Neste capítulo, vimos a importância de deixar as classes com código limpo e quais as principais técnicas para isso. No próximo capítulo, será apresentada a importantíssima regra do escoteiro.

Referências

O conteúdo completo da série **Design Principles and Design Patterns** (MARTIN, 2000) está disponível on-line em https://web.archive.org/web/20150906155800/http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf.

Existem excelentes livros em português (não são traduções) sobre SOLID:

1. ***Orientação a Objetos e SOLID para Ninjas — Projetando classes flexíveis***, de autoria de Mauricio Aniche (2015, Casa do Código);
2. ***Desbravando SOLID — Práticas avançadas para códigos de qualidade em Java moderno***, de autoria de Alexandre Aquiles (2022, Casa do Código).

Capítulo 10
A regra do escoteiro

João Kacau tinha uma grande preguiça de arrumar seu quarto. Sabe com qual frequência ele arrumava seu quarto? Bom, só quando ele demorava muito para encontrar alguma coisa ou quando nem ele aguentava mais o cheiro do local.

Então ele compartilhou sua angústia sobre seu quarto com amigos. Sorte a dele que um dos amigos lhe falou sobre uma metodologia/filosofia japonesa chamada **5S**. João Kacau ficou surpreso ao saber que ela foi criada na década de 1950, sendo muito usada até hoje em vários tipos de empresas, principalmente em fábricas. Esse nome se deu por ela ser baseada em 5 "sensos", conforme detalhado a seguir.

Os 5 Sensos do 5S

1. **Seiri** (Senso de **Utilização**): Separar e manter somente o necessário e descartar o que é desnecessário.
2. **Seiton** (Senso de **Organização** ou **Ordem**): Organizar o necessário. Uma vez que só o necessário está no ambiente, deixe ele organizado.
3. **Seiso** (Senso de **Limpeza**): Limpar o local. Sujou, limpou. Não acumule lixo, por menor que seja.
4. **Seiketsu** (Senso de **Padronização**): Padronizar as atividades. Padrões nos ajudam a não ficar recorrendo à memória, nos fazendo economizar tempo. Exemplo para um quarto: guardar os mesmos tipos de roupa no mesmo lugar.
5. **Shitsuke** (Senso de **Disciplina**): Manter o que foi feito. É preciso "não deixar a peteca cair" e repetir e manter os 4 sensos anteriores sempre, fazendo deles rotina. Os 4 sensos anteriores promovem a arrumação. O último permite que ela se mantenha.

Um resumo do funcionamento do 5S pode ser visto na figura a seguir.

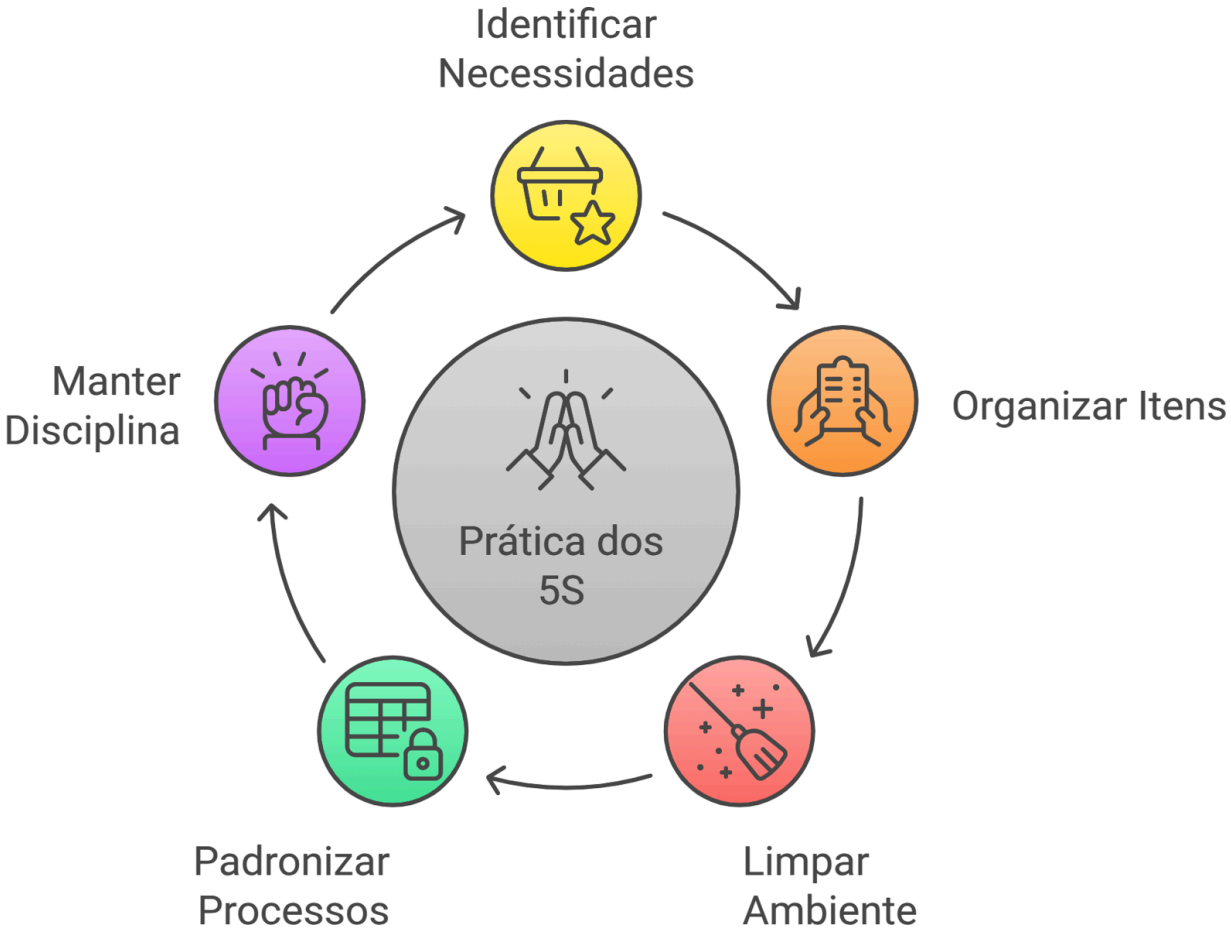


Figura 10.1: Funcionamento do 5S.

João Kacau curtiu a ideia e implantou o 5S para arrumar e manter arrumado seu quarto. Ademais, notou que, depois do trabalho inicial, a manutenção acaba não sendo tão trabalhosa. Quem visita seu quarto agora, não importa o dia e a hora, sempre vê um quarto arrumado, limpo e livre de mau cheiro. Coincidentemente (será?), o namoro dele que começou depois disso já dura 1 ano, sendo que os anteriores acabavam sempre depois da primeira visita da namorada à sua casa.

Quando um código e uma tarefa chegam até nós, podemos pensar que basta implementar boas práticas no código que vamos escrever, ignorando toda má prática observada no código já existente. Mas será que isso é o certo a fazer? Bom, segundo uma das práticas de Código Limpo, não! Estou falando da **Regra do escoteiro**. Seu

nome original é *"the boy scout rule"* e ela é baseada em um princípio do escotismo que diz: *"Uma vez que você acampa, você deve deixar a área onde está mais limpa do que quando a encontrou."* No contexto de programação, isso significa deixar o código mais limpo do que estava quando ele chegou até você.

Em geral é mais fácil criar um código bem escrito do zero do que melhorar um código que chega para a realização de uma tarefa. Isso porque, sempre que vamos lidar com código já existente, temos que entender o que o código se propõe a fazer. E é justamente um código que não é bom que é difícil e trabalhoso de entender.

Pode não ser a coisa mais empolgante e motivadora do mundo. Pense, porém, na nobreza que há em impedir que alguns trechos de código ruim continuem existindo num projeto. Ao fazer isso você vai, potencialmente:

- Evitar que bugs aconteçam
- Evitar que código ruim seja replicado por ser usado como base para novas funcionalidades do sistema
- Reduzir o **débito técnico**

> O que é **DÉBITO TÉCNICO**?
>
> No contexto de desenvolvimento de software, o débito técnico é como se fosse uma "dívida" que você acumula sempre que vai optando por soluções rápidas e pragmáticas em vez de investir em soluções mais robustas e sustentáveis de longo prazo. Em outras palavras, é o conjunto de todo o código que precisa ser refatorado (refeito de uma forma melhor).

Devo então refatorar (melhorar) todo o projeto sempre que mexer nele?

Não. Assim como o escoteiro não deve arrumar o acampamento inteiro, você não deve simplesmente refatorar um sistema inteiro por

causa de uma tarefa. O que deve ser melhorado é o código diretamente relacionado à sua tarefa.

Existe algum passo a passo para executar a regra do escoteiro?

Oficialmente, não. Porém, eu citei o 5S na história do início deste capítulo para oferecê-lo como sugestão de modelo a seguir quando for implantar a regra do escoteiro. Se você leu os capítulos anteriores, relembre deles por um instante e notará que quase todos os princípios e práticas do Código Limpo podem ser usados para executar o 5S. Uma sugestão de usar o 5S para executar a regra do escoteiro seria como a tabela a seguir:

Senso de	Objetivo	Prática/princípio de Código Limpo
Utilização	Separar o necessário do desnecessário	DRY (cap 3) Boas práticas de comentários (cap 4)
Organização ou Ordem	Organizar o necessário	Bons nomes (cap 2) Boas práticas de funções (cap 3) Formatação (cap 5) Boas práticas de classes (cap 9)
Limpeza	Limpar o local de trabalho	Tratamento de erros (cap 7) Boas práticas de comentários (cap 4) Formatação (cap 5)
Padronização	Padronizar as atividades	Bons nomes (cap 2) Boas práticas com Objetos (cap 6) Convenções de código (apêndice A)
Disciplina	Manter o que foi feito	Testes de unidade (cap 8) Uso de ferramentas (apêndice B)

Nunca se esqueça de que ter um código limpo traz inúmeros benefícios para você e para todos da equipe. Logo, se você simplesmente ignorar um código "não limpo", ao invés de melhorá-lo, você mesmo pode sofrer as consequências no futuro e é corresponsável pela continuidade do débito técnico do projeto.

Neste capítulo, vimos a importância de manter constante o código limpo e deixar o código mais limpo do que quando encontramos. A seguir, teremos alguns apêndices que complementam todas as técnicas ensinadas nesta obra.

Capítulo 11
Apêndice A — Convenções de código

Imagine um fogão: independente da marca, já vem à mente aquelas bocas e aqueles botões giratórios para regular a força do gás em cada boca, correto? Seria mais difícil, pelo menos por um tempo, operar um fogão fora desse padrão, não acha? Ou ainda: um teclado de computador no qual as posições das teclas são totalmente diferentes dos padrões comuns e do chamado *teclado mecânico*. Pelo menos por um tempo, fazer suas tarefas do cotidiano em um teclado assim levaria mais tempo, certo?

O mesmo ocorre com programas de computador: toda linguagem de programação possui **convenções**. Se um código-fonte não as seguir, provavelmente vai funcionar (afinal, convenções não são regras), mas será mais difícil de ser compreendido e manipulado (pelo menos por um tempo) por cada nova pessoa que tiver de fazer a manutenção dele. Neste apêndice, serão apresentados os **padrões de escrita de nomes** de componentes em um código-fonte e as **convenções de código** das linguagens de programação usadas nesta obra.

11.1 Padrões de escrita de nomes

Nas linguagens de programação, existem alguns padrões de nomes para componentes em um código-fonte (variáveis, classes, funções etc.), que são:

Snake case (snake_case)

Nesse padrão, todas as palavras ficam em **caixa baixa** e cada palavra é separada por um *underline* (_). Exemplos: `altura` ;

`data_nascimento`; `nome_sobrenome_solteiro`.

Macro case (MACRO_CASE)

Parecido com o *snake_case*, porém todas as palavras ficam em **caixa alta**. Também é citado com o nome ***Upper case*** (**UPPER_CASE**). Exemplos: `ALTURA`; `DATA_NASCIMENTO`; `NOME_SOBRENOME_SOLTEIRO`.

Camel case (camelCase)

Nesse padrão, toda primeira palavra deve estar em **caixa baixa**. As demais, se houver, devem ter apenas a **primeira letra maiúscula**. Não há qualquer caractere entre as palavras. Exemplos: `altura`; `dataNascimento`; `nomeSobrenomeSolteiro`.

Pascal case (PascalCase)

Muito parecido com o *camelCase*; a única diferença é que a primeira palavra inicia com **letra maiúscula**. Também é citado com o nome **StudlyCaps** Exemplos: `Altura`; `DataNascimento`; `NomeSobrenomeSolteiro`.

Kebab case (kebab-case)

Nesse padrão, todas as palavras ficam em **caixa baixa** e cada palavra é separada por um *hífen* (**-**). Esse, definitivamente, não possui uma unanimidade quanto ao nome: Também é citado com os nomes **caterpillar-case**, **lisp-case**, **css-case**, **dash-case**, **hyphen-case** e **spinal-case**. Exemplos: `altura`; `data-nascimento`; `nome-sobrenome-solteiro`.

Flat case (flatcase)

Nesse padrão, tudo é escrito em **caixa baixa** e não há nenhum caractere que separe as diferentes palavras. Exemplos: `altura`; `datanascimento`; `nomesobrenomesolteiro`.

11.2 Convenções da linguagem Java

Pacotes: flatcase, porém com cada subpacote separado por ponto ("**.**"). Pode haver *underline* (_), mas só quando for para substituir *hífen* (**-**) presente no domínio da empresa/organização. Exemplos: `br.com.codigolimpo`; `org.apache`; `com.eu_mesmo` (caso o domínio da empresa for *eu-mesmo.com*).

- **Classes**: PascalCase.
- **Interfaces**: PascalCase.
- **Atributos de instância**: camelCase.
- **Variáveis**: camelCase.
- **Métodos**: camelCase.
- **Construtores**: PascalCase.
- **Parâmetros de métodos e construtores**: camelCase.
- **Atributos de classe**: MACRO_CASE.
- **Enums**: MACRO_CASE.

Outras linguagens de programação que possuem convenções muito ou totalmente semelhantes às de Java: **Groovy**, **JavaScript**, **Kotlin** e **Scala**.

11.3 Convenções da linguagem Python

As convenções de nomes para Python podem ser encontradas em um documento chamado **PEP 8 -- Style Guide for Python Code**, que está resumido a seguir.

- **Pacotes**: snake_case.
- **Módulos**: snake_case.
- **Classes**: PascalCase.
- **Atributos de instância**: snake_case.
- **Variáveis globais**: snake_case.
- **Variáveis**: snake_case.

- **Parâmetros de métodos**: snake_case.
- **Primeiro parâmetro de métodos de instância**: A convenção é que tenha o nome `self`.
- **Primeiro parâmetro de métodos de classe**: A convenção é que tenha o nome `cls`.
- **Atributos de classe**: snake_case.
- **Funções**: snake_case.
- **Constantes**: MACRO_CASE.

Capítulo 12
Apêndice B — Ferramentas

Por mais que conheçamos as orientações de Código Limpo, sempre é bom ter uma mãozinha nos ajudando, certo? Por mais que tenhamos a reta intenção de manter o código limpo, somos humanos. Logo, ficamos cansados, estressados, tristes, aborrecidos etc. Por isso, um software para nos ajudar a manter o código do jeito que queremos é essencial no cinto de utilidades de qualquer profissional de programação.

Esse tipo de software são as ferramentas de **análise de qualidade de código**. Elas realizam algo chamado **análise estática de código** que recebe esse nome porque é feita de forma estática, ou seja, sem que o código esteja em execução. A intenção aqui **não** é ser um tutorial de uso desse tipo de ferramenta, o que seria conteúdo para um livro inteiro. Aqui será apenas descrito o seu conceito, falaremos sobre a sua importância e serão apresentadas algumas sugestões.

Essas ferramentas varrem um código-fonte à procura de erros e potenciais problemas. O que uma ferramenta desse tipo procura pode variar, mas, em geral, elas procuram por:

- Código que impeça a compilação/interpretação;
- Más práticas;
- Fuga de convenções da linguagem;
- Problemas potenciais de segurança;
- Pontos com possibilidade de erro de ponteiro nulo;
- Código repetitivo;
- Código que pode provocar redução considerável de desempenho.

Elas geram sempre um relatório, que pode ser pequeno e localizado, como no caso de plugins de IDE, ou grande e detalhado, como no

caso de quando são integrados a algum sistema de CI/CD.

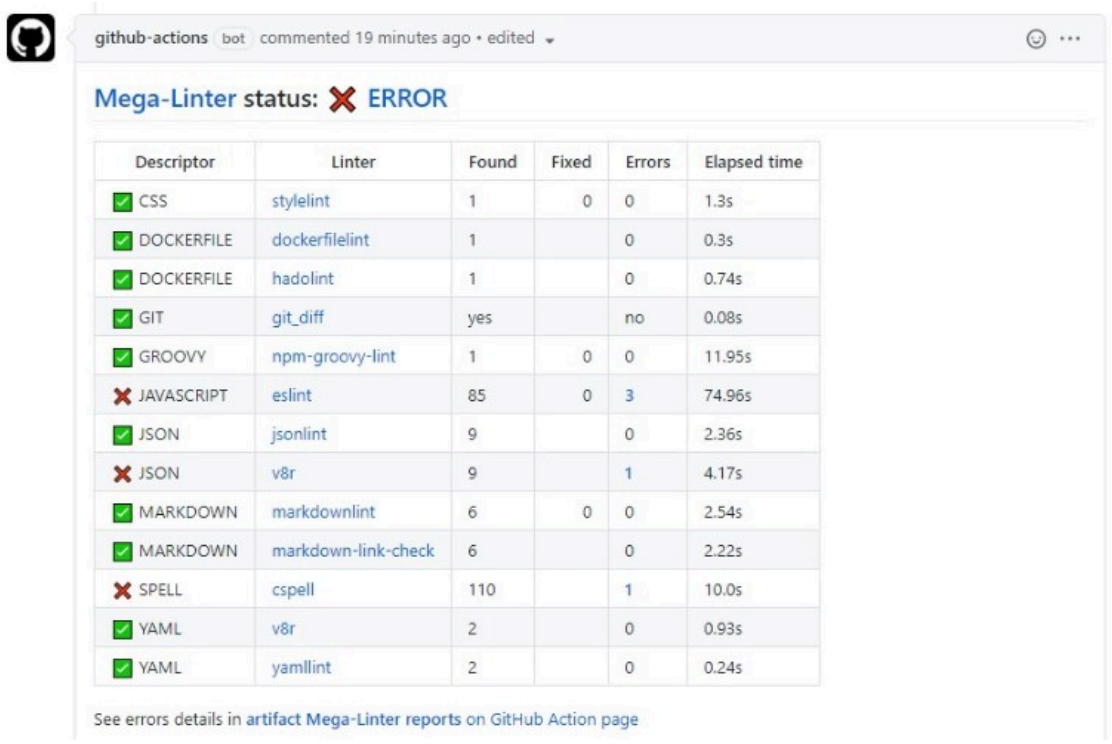
> **O que é Lint/Linter**
>
> É comum ver essas ferramentas de análise de qualidade de código sendo chamadas de **Lint** ou **Linter**. Esses termos são comumente encontrados escritos desse jeito mesmo em textos em português. Até existem traduções para **lint**, que seriam *"gaze"* ou *"fibra de algodão"*, mas que, definitivamente, não ficariam bem. O que soa melhor: "Ferramenta Lint"/"Ferramenta Linter" ou "ferramenta de gaze/fibra de algodão"? Esse termo foi usado para classificar esse tipo de programa pela primeira vez no utilitário **Lint**, criado por **Stephen C. Johnson** em 1978. Ele visava analisar códigos escritos na linguagem **C**. Portanto, não estranhe se vir pessoas usando um verbo como "*fazer lint*" do código. Várias ferramentas desse tipo têm o termo **Lint**/**Linter** no nome.

Algumas ferramentas de Lint são *plugins*/*extensões* de IDEs, o que facilita o seu uso; outras são utilitários que podem ser adicionados a *pipelines* de sistemas de CI/CD. Graças a isso, fica mais fácil para que tanto equipes de desenvolvimento quanto de DevOps/infra usem essas ferramentas, minimizando ao máximo problemas de código de má qualidade.

Como a criação (e defasagem) de ferramentas ocorre de forma um tanto rápida, pode ser que a lista nesta obra já esteja defasada no momento em que você a lê (a saber, este capítulo foi atualizado pela última vez em janeiro de 2023). Tenha apenas em mente que o uso desse tipo de ferramenta deve fazer parte de seu cotidiano no desenvolvimento de software, portanto procure sempre saber quais as melhores ferramentas da atualidade para isso. Algumas sugestões de ferramentas populares e que são boas opções tanto para Java quanto para Python são as citadas a seguir.

Mega-Linter

O Mega-Linter (https://megalinter.io) pode analisar projetos de mais de 14 linguagens de programação e possui mais de 70 *linters* (tipos de analisadores). Possui relatórios avançados que indicam em quais critérios de qualidade e/ou segurança o código não passou. Pode ser executado localmente (como CLI) ou em qualquer sistema CI/CD. Para uso local, requer a instalação do **Node.js** e **npx**, ou a instalação via imagem **Docker**. É possível ver uma imagem de um relatório gerado por ele na figura a seguir.



github-actions bot commented 19 minutes ago • edited

Mega-Linter status: ERROR

Descriptor	Linter	Found	Fixed	Errors	Elapsed time
CSS	stylelint	1	0	0	1.3s
DOCKERFILE	dockerfilelint	1		0	0.3s
DOCKERFILE	hadolint	1		0	0.74s
GIT	git_diff	yes		no	0.08s
GROOVY	npm-groovy-lint	1	0	0	11.95s
JAVASCRIPT	eslint	85	0	3	74.96s
JSON	jsonlint	9		0	2.36s
JSON	v8r	9		1	4.17s
MARKDOWN	markdownlint	6	0	0	2.54s
MARKDOWN	markdown-link-check	6		0	2.22s
SPELL	cspell	110		1	10.0s
YAML	v8r	2		0	0.93s
YAML	yamllint	2		0	0.24s

See errors details in artifact Mega-Linter reports on GitHub Action page

Figura 12.1: Relatório do Mega-Linter no GitHub Actions. Fonte: https://github.com/marketplace/actions/megalinter.

Semgrep

O Semgrep (https://semgrep.dev) tem suporte a mais de 17 linguagens de programação. Possui relatórios avançados. Pode ser executado localmente (como CLI) ou em qualquer sistema CI/CD. Para uso local, requer a instalação do **Python 3** ou a instalação via imagem **Docker**. É possível ver uma imagem de um relatório gerado por ele na figura a seguir.

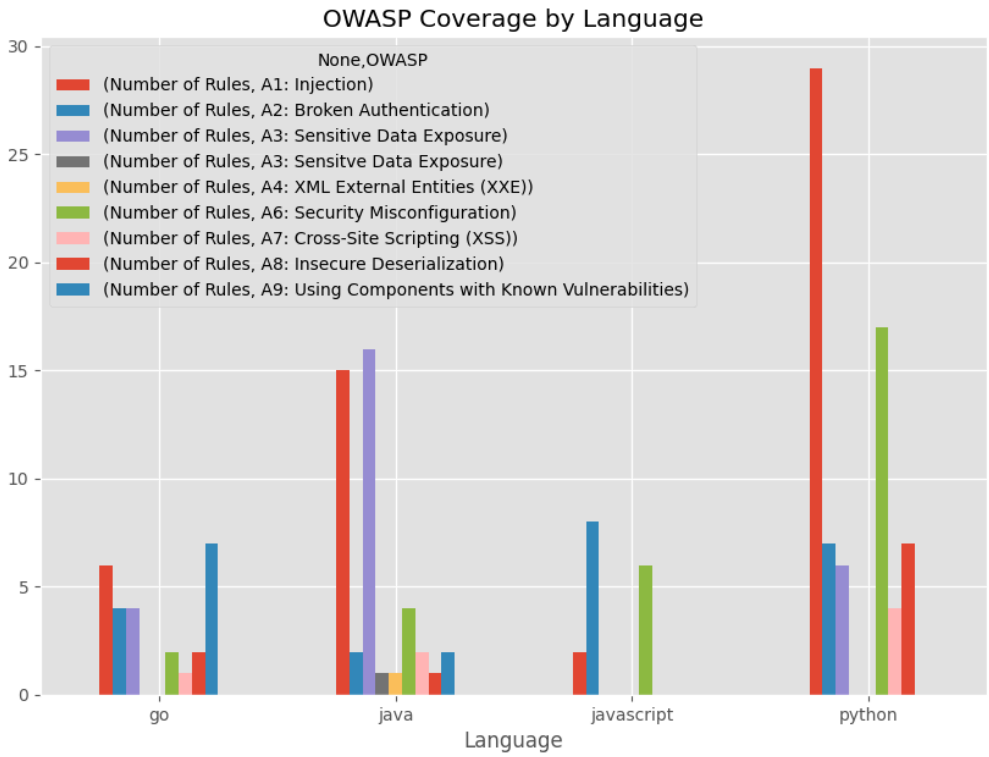


Figura 12.2: Relatório do Semgrep. Fonte: https://pypi.org/project/semgrep/0.15.0b1/.

SonarQube

O SonarQube (https://sonarqube.org) tem suporte a mais de 29 linguagens de programação. Possui relatórios avançados. Pode ser executado localmente (como plugin de IDE) ou em qualquer sistema CI/CD.

Para uso local, requer a instalação do plugin na IDE. Nesse caso, a ferramenta recebe o nome de **SonarLint** e possui versões para **IntelliJ Idea**, **Visual Studio Code**, **PyCharm**, entre outras. Para uso em sistemas CI/CD, a ferramenta se chama **SonarCloud**. É possível ver imagens de relatórios gerados pelo SonarLint e pelo SonarCloud nas figuras a seguir.

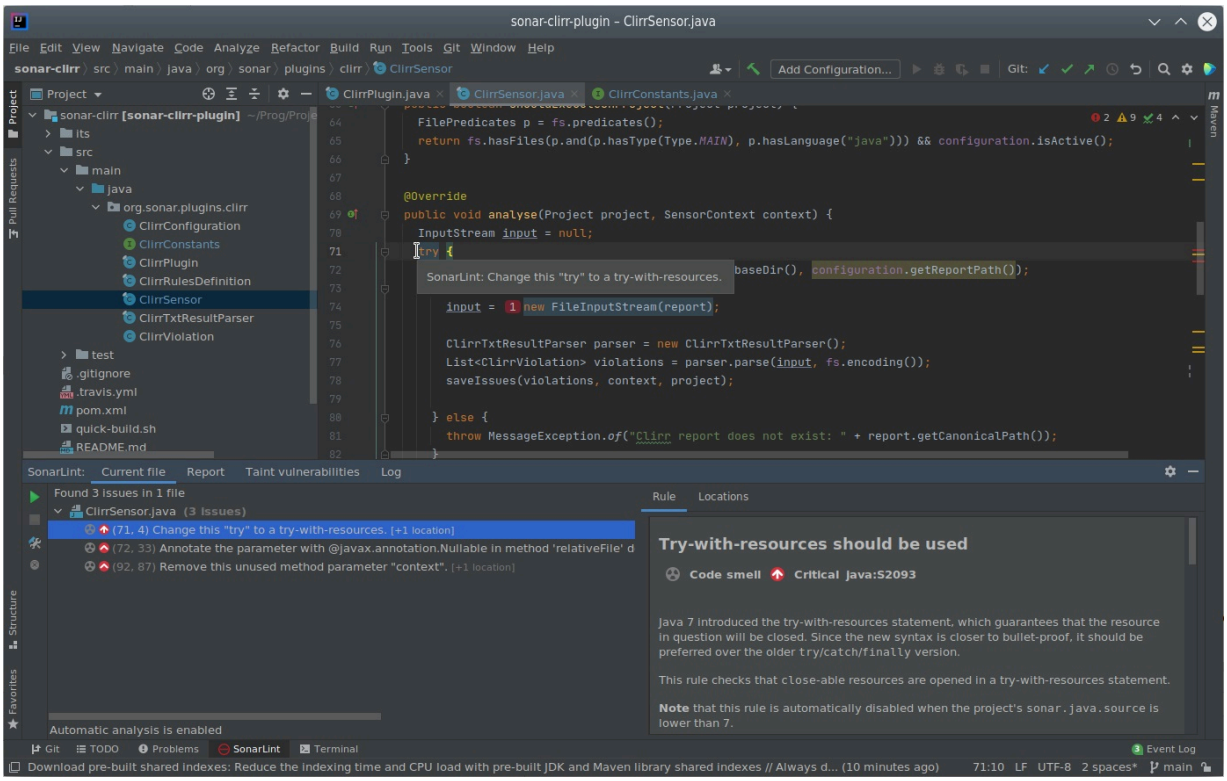


Figura 12.3: Relatório do SonarLint. Fonte: https://plugins.jetbrains.com/plugin/7973-sonarlint.

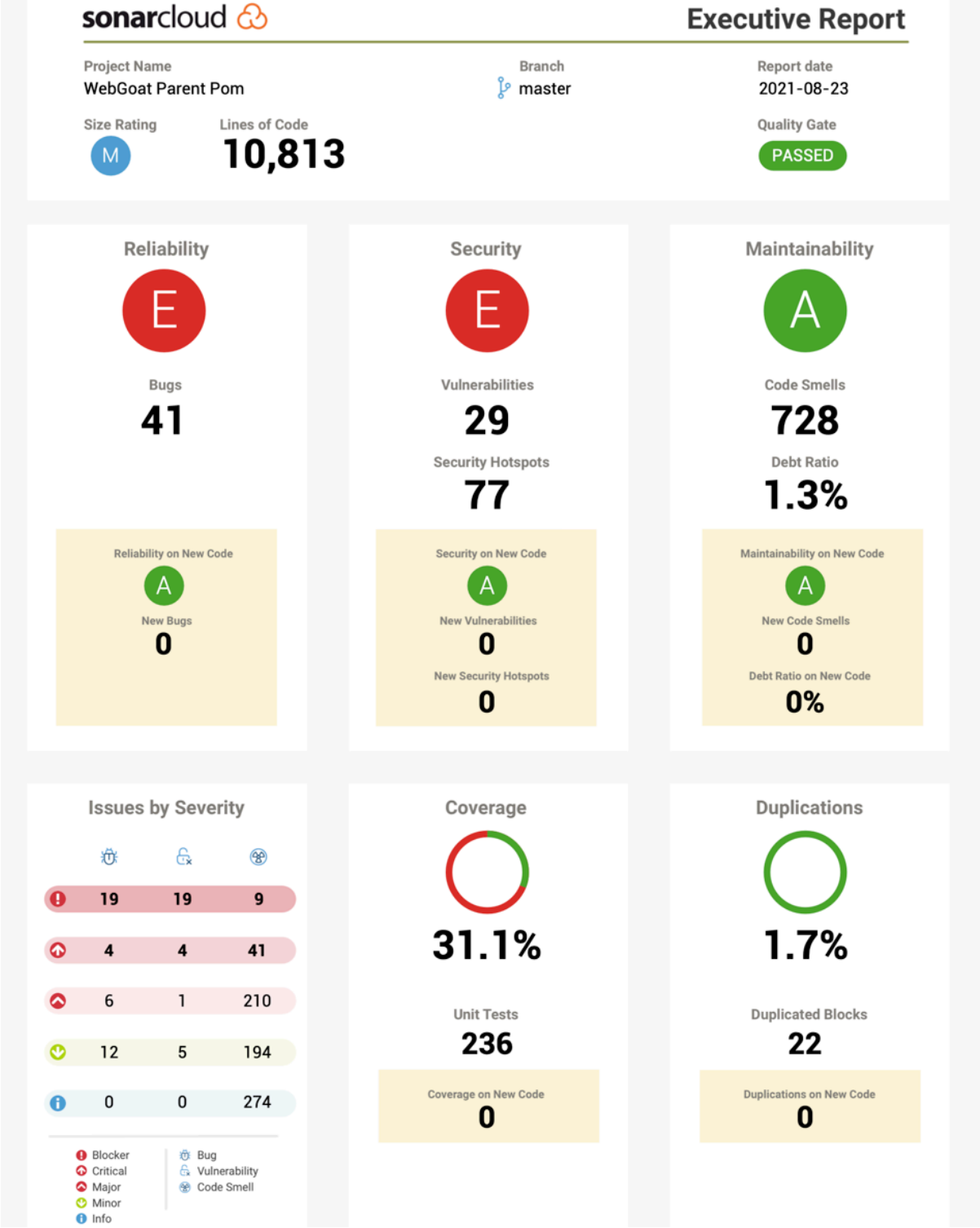


Figura 12.4: Relatório do SonarCloud. Fonte: https://www.bitegarden.com/how-dowload-pdf-report-sonarcloud.

Listagens atualizadas de ferramentas

Como já foi dito, ferramentas nascem e morrem muito rápido. Porém, há algumas iniciativas coletivas e abertas que visam manter listas atualizadas desses tipos de ferramenta. São elas:

- **Analysis Tools** (https://analysis-tools.dev/): ferramentas de análise estática para diversas linguagens. Código-fonte: https://github.com/mre/awesome-static-analysis.
- **Coleção de analisadores de código do GitHub**: https://github.com/collections/clean-code-linters.

Capítulo 13
Referências

ANICHE, Maurício. *Orientação a Objetos e SOLID para Ninjas*: Projetando classes flexíveis*. São Paulo: Casa do Código, 2015.*

BOCK, David. The Paperboy, The Wallet, and The Law Of Demeter. (s. d.). Disponível em: https://www2.ccs.neu.edu/research/demeter/demeter-method/LawOfDemeter/paper-boy/demeter.pdf. Acesso em: 9 fev. 2022.

BLOCH, Joshua. _Java Efetivo: As melhores práticas para a plataforma Java. 3. ed. Rio de Janeiro: Alta Books, 2019.

CRUZ, Felipe. *Python*: Escreva seus primeiros programas. São Paulo: Casa do Código, 2015.

DARWIN, Ian F. *Checking C programs with lint*. Sebastopol (EUA): O'Reilly Media, 1988.

DARWIN, Ian F. *Java Cookbook*: Problems and Solutions for Java Developers. 4. ed. Sebastopol (EUA): O'Reilly Media, 2020.

FARREL, Nick. Torvalds signs off on banning a list of racist words. *Fudzilla*, 2020. Disponível em: https://www.fudzilla.com/news/51167-torvalds-signs-off-on-banning-a-blacklist-of-racist-words. Acesso em: 9 jan. 2021.

FOWLER, Martin. POJO. *martinfowler.com*, 2003. Disponível em: https://www.martinfowler.com/bliki/POJO.html. Acesso em: 18 out. 2022.

GANESH, S. G. *60 Tips On Object Oriented Programming*. New York (EUA): McGraw-Hill Education, 2007.

GOODGER, David; VAN ROSSUM, Guido. PEP 257 - Docstring Conventions. *Python Enhancement Proposals*, 2001. Disponível em: https://www.python.org/dev/peps/pep-0257/. Acesso em: 10 dez. 2020.

GOOGLE. *Google Java Style Guide*. Disponível em: https://google.github.io/styleguide/javaguide.html. Acesso em: 1 ago. 2019.

GULATI, Shekhar; SHARMA, Rahul. *Java Unit Testing with JUnit 5*: Test-Driven Development with JUnit 5.

New York (EUA): Apress, 2017.

LANDAU, Elizabeth. Tech Confronts Its Use of the Labels 'Master' and 'Slave'. *Wired*, 2020. Disponível em https://www.wired.com/story/tech-confronts-use-labels-master-slave/. Acesso em: 9 jan. 2021.

LJUNG, Kevin; GONZALEZ-HUERTA, Javier. To Clean Code or Not to Clean Code: A Survey Among Practitioners. In: TAIBI, D. et al. (Eds.). Product-Focused Software Process Improvement 23nd International Conference - PROFES 2022. *Lecture Notes in Computer Science*, v. 13709, p. 298-315. Cham: Springer, 2022. Disponível em: https://doi.org/10.1007/978-3-031-21388-5_21. Acesso em 14 mar. 2023.

MARTIN, Robert C. *Clean Code*: A Handbook of Agile Software Craftsmanship. Upper Saddle River (EUA): Prentice Hall, 2008.

MARTIN, Robert C. *The Clean Architecture*. 2012. Disponível em: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html. Acesso em: 06 jan. 2025.

MARTIN, Robert C. *Design Principles and Design*

Patterns. 2000. Disponível em: http://staff.cs.utu.fi/~jounsmed/doos_06/material/DesignPrinciplesAndPatterns.pdf. Acesso em: 15 dez. 2019.

MELO, Ana Cristina. *Desenvolvendo Aplicações com UML 2.2*. Rio de Janeiro: Brasport, 2011.

MYERS, Rob. A History of Test-Driven Development (TDD), as Told in Quotes. *Agile For All*, 2020. Disponível em: https://agileforall.com/history-of-tdd-as-told-in-quotes/. Acesso em: 25 ago. 2022.

ORACLE. Naming a Package. In: The Java™ Tutorials. c2020. Disponível em: https://docs.oracle.com/javase/tutorial/java/package/namingpkgs.html. Acesso em: 20 mar. 2020.

ORACLE. 5 - Record Classes. In: Java Language Updates. c2023. Disponível em https://docs.oracle.com/en/java/javase/16/language/records.html. Acesso em: 10 fev. 2023.

ORACLE. *JavaBeans(TM) Specification 1.01 Final Release*. Disponível em https://download.oracle.com/otndocs/jcp/7224-javabeans-1.01-fr-spec-oth-JSpec/. Acesso em: 9

mar. 2023.

ORACLE BRASIL. *How to Write Doc Comments for the Javadoc Tool*. c2021. Disponível em https://www.oracle.com/br/technical-resources/articles/java/javadoc-tool.html. Acesso em: 11 de dezembro de 2021.

PYTHON.ORG. *unittest*: Unit testing framework. c2020. Disponível em: https://docs.python.org/3/library/unittest.html. Acesso em: 20 mar. 2020.

PYTHON.ORG. *Built-in Exceptions*. c2022. Disponível em: https://docs.python.org/3/library/exceptions.html. Acesso em: 11 jun. 2022.

SUN. *Java Code Conventions*. 1997. Disponível em https://www.oracle.com/technetwork/java/codeconventions-150003.pdf. Acesso em: 1 ago. 2020.

VAN ROSSUM, Guido; WARSAW, Barry; COGHLAN, Nick. PEP 8 – Style Guide for Python Code. *Python Enhancement Proposals*, 2001. Disponível em: https://www.python.org/dev/peps/pep-0008/. Acesso em: 2 ago. 2019.

WATSON, David; WILLIAMS, Helen. *Cambridge IGCSE and O Level Computer Science*. 2. ed. London (England): Hodder Education, 2021.

YOSHIRIRO, José. Proposta de ensino das técnicas de Código Limpo em IES de acordo com sua adoção na indústria de TI do Brasil. Dissertação (Mestrado em Ciências - Área: Administração das Organizações) - Faculdade de Economia, Administração e Contabilidade de Ribeirão Preto FEA-RP - Universidade de São Paulo, São Paulo, 2024.